DOCUMENT RESUME

ED 238 727 SE 043 748

AUTHOR	VanLehn, Kurt
TITLE	Felicity Conditions for Human Skill Acquisition: Validating an AI-Based Theory. Cognitive and Instructional Sciences Series.
INSTITUTION	Xerox Corp., Palo Alto, CA. Palo Alto Research Center.
SPONS AGENCY	Office of Naval Research, Arlington, Va. Personnel and Training Research Programs Office.
REPORT NO	CIS-21
PUB DATE	Nov 83
CONTRACT	N00014-82C-0067
NOTE	338p.
PUB TYPE	Reports - Research/Technical (143)
EDRS PRICE	MF01/PC14 Plus Postage.
DESCRIPTORS	Algebra; *Artificial Intelligence; Cognitive Development; *Cognitive Processes; Computer Programs; Elementary Secondary Education; Error Patterns; Fractions; Language Patterns; *Language Processing; Learning; *Learning Theories; Linguistic Performance; *Mathematics Education; Mathematics Instruction; *Skill Development; Subtraction
IDENTIFIERS	*Mathematics Education Research

ABSTRACT

A theory of how people learn certain procedural skills is presented. It is based on the idea that the teaching and learning that goes on in a classroom is like an ordinary conversation. The speaker (teacher) compresses a non-linear knowledge structure (the target procedure) into a linear sequence of utterances (lessons). The listener (student) constructs a knowledge structure (the learned procedure) from the utterance sequence (lesson sequence). Speakers unknowingly obey certain constraints, called felicity conditions or conversational postulates, which help listeners construct an appropriate knowledge structure. This research has shown that there are felicity conditions on lesson sequences that help students learn procedures. Three felicity conditions were discovered, forming the central hypotheses in the learning theory, which was embedded in a model, a large computer program using artificial intelligence (AI) techniques. The model's performance was compared to data from several thousand students learning to subtract multidigit numbers, add fractions, and solve simple algebraic equations. A key criterion for the theory is that the set of procedures that the model "learns" should exactly match the set of procedures that students actually acquire. Both testing the predictions and arguing for the validity of the theory are included in the report. (Author/MNS)

****
* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *
****

Palo Alto Research Centers

U.S. DEPARTMENT OF EDUCATION
NATIONAL INSTITUTE OF EDUCATION
EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)

☑ This document has been reproduced as received from the person or organization originating it.

☐ Minor changes have been made to improve reproduction quality.

- Points of view or opinions stated in this document do not necessarily represent official NIE position or policy.

ED238727

Felicity Conditions for Human Skill Acquisition: Validating an AI-Based Theory

Kurt VanLehn

"PERMISSION TO REPRODUCE THIS MATERIAL HAS BEEN GRANTED BY

U.S. Government

TO THE EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)."

XEROX

REPORT DOCUMENTATION PAGE		**READ INSTRUCTIONS BEFORE COMPLETING FORM**
1. REPORT NUMBER CIS-21	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (*and Subtitle*) Felicity Conditions for Human Skill Acquisition: Validating an AI-based Theory		5. TYPE OF REPORT & PERIOD COVERED Interim Report
		6. PERFORMING ORG. REPORT NUMBER P8300048
7. AUTHOR(*s*) Kurt VanLehn		8. CONTRACT OR GRANT NUMBER(*s*) N00014-82C-0067
9. PERFORMING ORGANIZATION NAME AND ADDRESS Cognitive and Instructional Sciences Group Xerox Corporation/Palo Alto Research Center 3333 Coyote Hill Road, Palo Alto, CA 94304		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS NR 667-477
11. CONTROLLING OFFICE NAME AND ADDRESS Personnel and Training Research Programs Office of Naval Research (code 458) Arlington, VA 22217		12. REPORT DATE November 1983
		13. NUMBER OF PAGES 335
14. MONITORING AGENCY NAME & ADDRESS(*if different from Controlling Office*)		15. SECURITY CLASS. (*of this report*) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (*of this Report*) Approved for public release; distribution unlimited		
17. DISTRIBUTION STATEMENT (*of the abstract entered in Block 20, if different from Report*)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (*Continue on reverse side if necessary and identify by block number*) Cognitive Science, Learning, Artificial Intelligence, Skill Acquisition, Human Cognition, Arithmetic		
20. ABSTRACT (*Continue on reverse side if necessary and identify by block number*) See reverse side.		

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

Abstract

A theory of how people learn certain procedural skills is presented. It is based on the idea that the teaching and learning that goes on in a classroom is like an ordinary conversation. The speaker (teacher) compresses a non-linear knowledge structure (the target procedure) into a linear sequence of utterances (lessons). The listener (student) constructs a knowledge structure (the learned procedure) from the utterance sequence (lesson sequence). In recent years, linguists have discovered that speakers unknowingly obey certain constraints on the sequential form of their utterances. Apparently, these tacit conventions, called felicity conditions or conversational postulates, help listeners construct an appropriate knowledge structure from the utterance sequence. The analogy between conversations and classrooms suggests that there might be felicity conditions on lesson sequences that help students learn procedures. This research has shown that there are. For the particular kind of skill acquisition studied here, three felicity conditions were discovered. They are the central hypotheses in the learning theory. The theory has been embedded in a model, a large computer program that uses artificial intelligence (AI) techniques. The model's performance has been compared to data from several thousand students learning ordinary mathematical procedures: subtracting multidigit numbers, adding fractions and solving simple algebraic equations. A key criterion for the theory is that the set of procedures that the model "learns" should exactly match the set of procedures that students actually acquire, including their "buggy" procedures. However, much more is need for psychological validation of this theory, or any complex AI-based theory, than merely testing its predictions. Part of the research has involved finding ways to argue for the validity of the theory.

Felicity Conditions for Human Skill Acquisition: Validating an AI-based Theory

Kurt VanLehn

November 1983

Cognitive and Instructional Sciences Series
CIS-21

Corporate Accession P8300048

© Copyright Kurt VanLehn 1983.

The author hereby grants to M.I.T. and to Xerox Corporation permission to reproduce and to distribute copies of this document in whole or in part.

XEROX

PALO ALTO RESEARCH CENTER
3333 Coyote Hill Road / Palo Alto /
California 94304

Approved for public release;
Distribution unlimited.

Felicity Conditions for Human Skill Acquisition: Validating an AI-based Theory

Kurt VanLehn

Abstract

A theory of how people learn certain procedural skills is presented. It is based on the idea that the teaching and learning that goes on in a classroom is like an ordinary conversation. The speaker (teacher) compresses a non-linear knowledge structure (the target procedure) into a linear sequence of utterances (lessons). The listener (student) constructs a knowledge structure (the learned procedure) from the utterance sequence (lesson sequence). In recent years, linguists have discovered that speakers unknowingly obey certain constraints on the sequential form of their utterances. Apparently, these tacit conventions, called felicity conditions or conversational postulates, help listeners construct an appropriate knowledge structure from the utterance sequence. The analogy between conversations and classrooms suggests that there might be felicity conditions on lesson sequences that help students learn procedures. This research has shown that there are. For the particular kind of skill acquisition studied here, three felicity conditions were discovered. They are the central hypotheses in the learning theory. The theory has been embedded in a model, a large computer program that uses artificial intelligence (AI) techniques. The model's performance has been compared to data from several thousand students learning ordinary mathematical procedures: subtracting multidigit numbers, adding fractions and solving simple algebraic equations. A key criterion for the theory is that the set of procedures that the model "learns" should exactly match the set of procedures that students actually acquire, including their "buggy" procedures. However, much more is need for psychological validation of this theory, or any complex AI-based theory, than merely testing its predictions. Part of the research has involved finding ways to argue for the validity of the theory.

Acknowledgments

I would like to express my thanks to the people who encouraged and aided me in this work: John Seely Brown, for providing me with ten new ideas every time I talked with him, and for supporting me so generously for so many years; Hal Abelson, for helping me turn a shambles of arguments into a presentable piece of work, and for moral support at the critical times; Patrick Winston, for opening doors that I feared were closed, and for reading far too many pages; Andy DiSessa, for help in the clutch; Johan deKleer, for reading unreadable first drafts, and (incredibly) finding the holes; Richard Burton, for Lisp support, for Debuggy, for windsurfing, and for onion theory; Jamesine Friend, for the Southbay study and a dash of reality in the ivory tower; Elizabeth Berg, for cheerfully coding the data; Joan Bresnan, Haj Ross, Mitch Marcus, Candy Sidner: for permanently warping me in just the right ways; Jim Greeno, for early support and a proper scientific perspective; Peter Andrea and Stu Card, for thorough critiques of an early draft; Steve Purcell, for an early reading, not to mention the John Muir trail and constant encouragement; Betsey Summers, for loving support amid years of receding deadlines.

This research was supported by the Personnel and Training Research Programs, Psychological Sciences Division, Office of Naval Research, under Contract Number N00014-82C-0067, Contract Authority Identification Number NR 667-477. Reproduction in whole or in part is permitted for any purpose of the United States Government. Approved for public release; distribution unlimited.

TABLE OF CONTENTS

1. Objectives of the Research 6
1.1 The psychological goal: Step theory and repair theory 6
1.2 The methodological goal: competitive arguments for each hypothesis 11
1.3 Overview of the theory and the document 17

2. Sierra, the Model 21
2.1 The top level of Sierra 22
2.2 The representation of observables: lessons and diagnostic tests 26
2.3 The representation of procedures 31
2.4 The representation of grammars 34
2.5 The representation of patterns 37
2.6 An introduction to induction 39
2.7 The learner 43
2.8 Core procedure trees for the Southbay experiment 51
2.9 The solver 55
2.10 Observational adequacy 61
2.11 A comparison with other generative theories of bugs 68

PART 1: ARCHITECTURE LEVEL

3. Getting Started 72
3.1 Teleology or program? 72
3.2 What kind of learning goes on in the classroom? 75
3.3 Learning by discovery 79
3.4 Learning by analogy 80
3.5 Learning by being told 82
3.6 Summary and formalization 83

4. The Disjunction Problem 86
4.1 An introduction to the disjunction problem 86
4.2 Barring disjunction from procedures 92
4.3 Neves' ALEX learner 94
4.4 Exactly one disjunct per lesson 95
4.5 Minimal disjuncts vs. one-disjunct-per-lesson 96
4.6 Formal hypotheses 99

5. The Invisible Objects Problem 101
5.1 Barring invisible objects 104
5.2 Unbiased induction with lots of examples 104
5.3 Minimal number of invisible object designators 105
5.4 Show work 107
5.5 Invisible objects, disjunctions, and Occam's Razor 110

6. Local Problem Solving 111
6.1 Explaining bugs with overgeneralization 111
6.2 Stretching overgeneralization to account for certain bugs 115
6.3 Impasse-repair independence 117
6.4 Dynamic vs. static local problem solving 122
6.5 Formal hypotheses 127

7. Deletion 129
7.1 Local problem solving will not generate certain bugs 129
7.2 Overgeneralization should not generate the deletion bugs 130
7.3 The problems of defining a deletion operator 131

8. Summary: Architecture Level 132
8.1 The architecture of the model 132
8.2 Hypotheses and their support 133
8.3 Commentary on the arguments and inherent problems 137
8.4 Preview of Part 2 138

PART 2: REPRESENTATION LEVEL

9. Control Flow 141
9.1 Chronological, Dependency-directed and Hierarchical Backup 142
9.2 One-disjunct-per-lesson entails recursive control structure 144
9.3 More powerful control structures 147
9.4 Summary and formalization 147

10. Goal Types 152
10.1 Assimilation is incompatible with the And hypothesis 154
10.2 The And-Or hypothesis simplifies the Backup repair 156
10.3 Rule deletion needs the And-or distinction 156
10.4 Satisfaction Conditions 160
10.5 Summary, formal hypotheses and conflict resolution 163

11. Data Flow 167
11.1 The hypothesis that there is no data flow 168
11.2 Focus is not globally bound 169
11.3 The applicative hypothesis 172
11.4 Summary and formalization 173

12. Searching the Current Problem State 175
12.1 Search loops 175
12.2 Pattern matching 176
12.3 Function nests as representations of paths 177
12.4 Summary and formalization 178

13. Ontology 180
13.1 Problem state spaces 181
13.2 Aggregate object definitions: fixed spatial relations 183
13.3 Grammars 185
13.4 Summary and formal hypotheses 188

14. Pattern Power 191
14.1 Viewing patterns as logics 191
14.2 Predicates, conjunctions and Skolem constants 193
14.3 Constants, negations and functions 193
14.4 Disjunctions are needed for variables and Skolem functions 194
14.5 Disjunctions in patterns 195
14.6 Summary and formal hypotheses 197

15. Syntax of the Representation Languages 198
15.1 Syntax of the procedure representation language .198
15.2 Syntax of the grammar representation language 202

16. Summary: Representation Level 206
16.1 The interface issue 207
16.2 The internal state issue 209
16.3 The learner 211
16.4 The solver 214
16.5 Preview of the bias level 219

PART 3: BIAS LEVEL

17. Two Patterns or One 220
17.1 The fetch bugs 220
17.2 Test patterns ≠ fetch patterns 223
17.3 Formal hypotheses 224

18. Fetch Patterns 225
18.1 Version spaces 225
18.2 Two hypotheses 227
18.3 The pattern focus hypothesis 228
18.4 Teleological rationalizations 230
18.5 Exact matching vs. Closest matching 231
18.6 When does mismatching trigger repair? 233
18.7 Summary 233

19. Test Patterns and Skeletons 234
19.1 The topological bias hypothesis 235
19.2 Step Schemata 243
19.3 Summary 247

20. Summary: Bias level 248

21. Conclusions 254
21.1 Strengths and weaknesses in the theory 254
21.2 Directions for future research 256

References 258

Appendices

A1 A bug glossary 266
A2 Observed bug sets 273
A3 Bug occurrence frequencies 277
A4 Predicted bug sets 281
A5 Observed bug sets, overlapped by predicted bug sets 284
A6 Predicted bug sets, overlapped by observed bug sets 289
A7 Version spaces as applicability conditions 293
A8 Interstate reference 297
A9 The Backup repair and the goal stack 303
A10 Satisfaction conditions 319

Chapter 1
Objectives of the Research

There are two goals for the research presented here. One is psychological and the other is methodological. The psychological goal is to formulate and validate a theory of a certain kind of human learning. The methodological goal is to use artificial intelligence (AI) techniques to model that learning, and to do it in such a way that the complexity of the AI-based model does not prevent the theory from meeting rigorous criteria of scientific validity. The first section of this chapter discusses the psychological goal; the second section discusses the methodological one. The third section introduces the organization of the rest of the document.

1.1 The psychological goal: Step theory and repair theory

One goal of this research is a psychologically valid theory of how people learn certain procedural skills. There are other AI-based theories of skill acquisition (e.g., Anderson, 1982; Newell & Rosenbloom, 1981). However, their objectives differ from the ones pursued here. They concentrate on *knowledge compilation*: the transformation of slow, stumbling performance into performance that is "faster and more judicious in choice" (Anderson, 1982, pg. 404). They study skills that are taught in a simple way: first the task is explained, then it is practiced until proficiency is attained. For instance, Anzai and Simon (1979) modelled a subject whose skill at solving the Tower of Hanoi puzzle evolved from a slow, stumbling first attempt into an ability to solve the puzzle rapidly using the optimal sequence of moves. The subject received no instruction after the initial description of the puzzle's operations and objectives. The research presented here studies skills that are taught in a more complex way: the instruction is a lesson sequence, where each lesson consists of explanation and practice of some small piece (subskill) of the ultimate procedural skill. Studying multi-lesson skill acquisition shifts the central focus away from practice effects (knowledge compilation) and towards a kind of student cognition that could be called *knowledge integration:* the construction of a procedural skill from lessons on its subskills.

This study puts more emphasis on the teacher's role than the knowledge compilation research does. It is not the case that multi-lesson skill acquisition occurs with just any lesson sequence. Rather, the lesson sequences are designed by the teacher to facilitate knowledge integration. Knowledge integration, in turn, is "designed" to work only with certain kinds of lesson sequences. So, what is really being studied is a teacher-student system that has both cognitive and cultural roots. An equally appropriate name for the central focus of this research is *knowledge communication:* the transmission of a procedural skill via lessons on its subskills.

The skills chosen for the present investigation are ordinary, written mathematical calculations. The main advantage of mathematical procedures, from the experimenter's point of view, is that they are virtually meaningless for the learner. They seem as isolated from common sense intuitions as the nonsense syllables of early learning research. In the case of the subtraction procedure, for example, most elementary school students have only a dim conception of its underlying semantics, which is rooted in the base-ten representation of numbers. When compared to the procedures they use to operate vending machines or play games, arithmetic procedures are as dry, formal and disconnected from everyday interests as nonsense syllables are different from real words. This isolation is the bane of teachers, but a boon to psychologists. It allows psychologists to study a skill that is much more complex than recalling nonsense syllables, and yet it avoids bringing in a whole world's worth of associations.

It is worth a moment to review how mathematical procedures are taught in a typical American school. In the case of subtraction, there are about ten lessons in its lesson sequence. The lesson sequence introduces the procedure incrementally, one step per lesson, so to speak. For instance, the first lesson might show how to do subtraction of two-column problems. The second lesson demonstrates three-column problem solving. The third introduces borrowing, and so on. The ten lessons are spread over about three years, starting in the late second grade (i.e. at about age seven). These lessons are interleaved with review lessons and lessons on many other topics. In the classroom, a typical lesson lasts an hour. The teacher solves some problems on the board with the class, then the students solve problems on their own. If they need help, they ask the teacher, or they refer to worked examples in the textbook. A textbook example consists of a sequence of captioned "shapshots" of a problem being solved, e.g.,

Take a ten to make 10 ones.	Subtract the ones.	Subtract the tens.
$\begin{array}{r} \overset{2}{\cancel{3}}{}^{1}5 \\ -\ 1\ 9 \\ \hline \end{array}$	$\begin{array}{r} \overset{2}{\cancel{3}}{}^{1}5 \\ -\ 1\ 9 \\ \hline 6 \end{array}$	$\begin{array}{r} \overset{2}{\cancel{3}}{}^{1}5 \\ -\ 1\ 9 \\ \hline 1\ 6 \end{array}$

Textbooks have very little text explaining the procedure (young children do not read well). Textbooks contain mostly examples and exercises.

Math bugs reveal the learning process

Error data are used in testing the theory. There have been many empirical studies of the errors that students make in arithmetic (Buswell, 1926; Brueckner, 1930; Brownell, 1941; Roberts, 1968; Lankford, 1972; Cox, 1975; Ashlock, 1976). A common analytic notion is to separate systematic errors from careless errors. Systematic errors appear to stem from consistent application of a faulty method, algorithm or rule. These errors occur along with the familiar unsystematic or "careless" errors (e.g., a facts error, such as $7-3=5$), or *slips* as I prefer to call them (c.f. Norman 1981). Since slips occur in expert performance as well as student behavior, the common opinion is that they are performance phenomena, an inherent part of the "noise" of the human information processor. Systematic errors on the other hand are taken as stemming from mistaken or missing knowledge about the skill, the product of incomplete or misguided learning. Only systematic errors are used in testing the present theory.

Brown and Burton used the metaphor of bugs in computer programs in developing a precise, detailed descriptive formalism for systematic errors (Brown & Burton, 1978). The basic idea is that a student's errors can be accurately reproduced by taking some formal representation of a correct procedure and making one or more small perturbations to it, e.g., deleting a rule. The perturbations are called bugs. A systematic error is represented by a set of one or more bugs in a correct algorithm for the skill. Bugs describe systematic errors with unprecedented precision. If a student makes no slips, then his or her answers on a test will be *exactly* matched by the buggy algorithm's answers, digit for digit. To illustrate the notion of bugs, consider the following problems, which display a systematic error:

$$\begin{array}{rrrrrrrr} 306 & 80 & 183 & 702 & 3005 & 7002 & 34 & 251 \\ -\ 138 & -\ 4 & -\ 95 & -\ 11 & -\ 28 & -\ 239 & -\ 14 & -\ 47 \\ \hline 78 & 76 & 88 & 591 & 1087 & 4873 & 24 & 244 \end{array}$$

One could vaguely describe these problems as coming from a student having trouble with borrowing, especially in the presence of zeros. More precisely, the student misses all the problems that require borrowing from zero. One could say that the student has not mastered the subskill of borrowing across zero. This description of the systematic error is fine at one level: it is a testable prediction about what new problems the student will get wrong. It predicts for example that the student will miss 305-117 and will get 315-117 correct. Systematic errors described at this level are the data upon which several psychological and pedagogical theories have been built (e.g., Durnin & Scandura, 1977).

Bugs go beyond describing what *kinds* of exercises the student misses. They describe the actual answers given. The student whose work appears above has a bug called Borrow-Across-Zero. A correct subtraction procedure has been perturbed by deleting the step wherein the zero is changed to a nine during borrowing across zero. This modification creates a procedure for answering subtraction problems. As a hypothesis, it predicts not only which new problems the student will miss, but also what the answers will be. For example, it predicts that the student above would answer 305-117=98 and 315-117=198. Since the bug-based descriptions of systematic errors predict behavior at a finer level of detail than missing-subskill descriptions, they have the potential to form a better basis for cognitive theories of learning and problem solving. Bug-based analysis is used in testing this theory.

It is often the case that a student has more than one bug at the same time. Indeed, the example given above illustrates co-occurrence of bugs. The last two problems are answered incorrectly but the bug Borrow-Across-Zero does not predict their answers (it predicts the two problems would be answered correctly). A second bug, called Diff-N−N=N, is present. When the student comes to subtract a column where the top and bottom digits are equal, instead of writing zero in the answer, the student writes the digit that appears in the column. So the student has two bugs at once.

Burton developed an automated data analysis program, called Debuggy (Burton, 1981). Using it, data from thousands of students learning subtraction were analyzed, and 76 different kinds of bugs were observed. Similar studies discovered 68 bugs in addition of fractions (Shaw et. al., 1982), several dozen bugs in simple linear equation solving (Sleeman, forthcoming), and 57 bugs in addition and subtraction of signed numbers (Tatsuoka & Baillie, 1982).

It is important to stress that bugs are only a notation for systematic errors and not an explanation. The connotations of "bugs" in the computer programming sense do not necessarily apply. In particular, bugs in human procedures are unstable. They appear and disappear over short periods of time, often with no intervening instruction, and sometimes even in the middle of a testing session (VanLehn, 1981; Bunderson, 1981). Often, one bug is replaced by another, a phenomenon called *bug migration*.

Mysteries abound in the bug data. Why are there so many different bugs? What causes them? What causes them to migrate or disappear? Why do certain bugs migrate only into certain other bugs? Often a student has more than one bug at a time — why do certain bugs almost always occur together? Do co-occurring bugs have the same cause? Most importantly, how is the educational process involved in the development of bugs?

This research was launched partly in order to explain the mysteries just mentioned. The goal is to give a unified account of what causes students to have just the specific bugs that they do have. As an illustration of the kind of explanations that the present theory offers, consider a common bug among subtraction students: the student always borrows from the *leftmost* column in the problem no matter which column originates the borrowing. Problem *a* below shows the correct placement of borrow's decrement. Problem *b* shows the bug's placement.

$$\text{a.}\quad \begin{array}{r} 3\,\overset{5}{6}\,{}^{1}5 \\ -\,1\,0\,9 \\ \hline 2\,5\,6 \end{array} \qquad \text{b.}\quad \begin{array}{r} \overset{2}{3}\,6\,{}^{1}5 \\ -\,1\,0\,9 \\ \hline 1\,6\,6 \end{array} \qquad \text{c.}\quad \begin{array}{r} \overset{5}{6}\,{}^{1}5 \\ -\,1\,9 \\ \hline 4\,6 \end{array}$$

(The small numbers represent the student's scratch marks.) Debuggy's name for this bug is Always-Borrows-Left. It is moderately common: In a sample of 375 students with bugs, six students had this bug. It has been observed for years (c.f. Buswell, 1926, pg. 173, bad habit number s27). However, no one has offered an explanation for why students have it. The theory offers the following explanation, which is based on the hypothesis that students use induction (generalization of examples) to learn where to place the borrow's decrement. Every subtraction curriculum that I know of introduces borrowing using only two-column problems, such as problem *c* above. Multi-column problems, such as *a*, are not used. Consequently, the student has insufficient information to unambiguously induce where to place borrow's decrement. The correct placement is in the left-adjacent column, as in *a*. However, two-column examples are also consistent with decrementing the leftmost column, as in *b*. If the student chooses the leftmost-column generalization, the student acquires Always-Borrow-Left rather than the correct procedure. According to this explanation, the cause of the bug is twofold: (1) insufficiently variegated instruction, and (2) an unlucky choice by the student.

The bugs that students exhibit are important data for developing the theory. These bugs will be called *observed* bugs. Equally important are bugs that students *don't* exhibit. When there are strong reasons to believe that a bug will never occur, it is called a *star* bug (after the linguistic convention of placing a star before sentences that native speakers would never utter naturally). Star bugs, and star data in general, are not as objectively attainable as ordinary data (VanLehn, Brown & Greeno, in press). But they are quite useful. To see this, consider again the students who are taught borrowing on two column problems, such as problem *c* above. In two-column problems, the borrow's decrement is always in the *tens* column. Hence "tens column" is an inductively valid description of where to decrement. However, choosing "tens column" for the decrement's description predicts that the student would place the decrement in the tens column regardless of where the borrow originates. This leads to strange solutions, such as *d* and *e* below:

$$\text{d.}\quad \begin{array}{r} 1\,{}^{1}5\,\overset{5}{6}\,5 \\ -\quad 9\,1\,0 \\ \hline 1\,6\,5\,5 \end{array} \qquad \text{e.}\quad \begin{array}{r} 3\,{}^{1}\overset{15}{6}\,5 \\ -\,1\,9\,0 \\ \hline 2\,6\,5 \end{array}$$

To my knowledge, this kind of problem solving has never been observed. In the opinion of several expert diagnosticians, it never will be observed. Always decrementing the tens column is a star bug. The theory should not predict its occurrence. This has important implications for the theory. The theory must explain why certain inductively valid abstractions (e.g., leftmost column) are used by students while certain other abstractions (e.g., tens column) are not.

These examples have illustrated one side of the research problem: to understand certain aspects of skill acquisition (i.e., knowledge integration/communication) by studying bugs. The next subsection is a brief discussion of the theory. It concentrates on the insights that have been obtained into how buggy procedures are acquired.

Step theory, repair theory and felicity conditions

For historical and other reasons it is best to view the present theory as an integration of two theories. *Step theory* describes how students acquire procedures from instruction. *Repair theory* describes how students barge through situations where their procedure has reached an impasse.* The two theories share the same representations of knowledge and much else. I will continue to refer to them together as "the theory."

Repair theory is based on the insight that students do not treat procedures as hard and fast algorithms. If they are unsuccessful in an attempt to apply a procedure to a problem, they are not apt to just quit, as a computer program does. Instead, they will be inventive, invoking certain general purpose tactics to change their current process state in such a way that they can continue the procedure. These tactics are simple ones, such as skipping an operation that can't be performed or backing up in the procedure and taking another path. Such local problem solving tactics are called *repairs* because they fix the problem of being stuck. They do not fix the underlying cause of the impasse. Given a similar exercise later, the student will reach the same impasse. On this occasion, the student might apply a different repair. This shifting among repairs is one explanation of bug migration. A remarkable early success of repair theory was predicting the existence of this kind of bug migration before the phenomenon was observed in the data.

Step theory is based on the insight that classroom learning is like a conversation in that there are certain implicit conventional expectations, called *felicity conditions*, that facilitate information transmission. In this domain, the felicity conditions all seem to reflect a single basic idea: students expect that the teacher will introduce just one new "piece" of the procedure per lesson, and that such "pieces" will be "simple" in certain ways. Although students do not have strong expectations about *what* procedures will be taught, they have strong expectations about *how* procedures will be taught. Step theory takes its name from a slogan that expresses the students' expectations: procedures are taught *one simple step at a time*. Several felicity conditions have been discovered, including:

1. Students expect a lesson to introduce at most one new "piece" of procedure that is, roughly speaking, one disjunct of a disjunction. Such "pieces" are called *subprocedures*. This felicity condition will be described in more detail in a moment.
2. Students induce their new subprocedure from examples and exercises. That is, students expect the lesson's material to correctly exemplify the lesson's target subprocedure.
3. The students expect the lesson to "show all the work" of the target subprocedure. This felicity condition, called the show-work principle, requires a little more explanation. Suppose a target subprocedure will ultimately involve holding some intermediate result mentally, as when solving 3+4+5, one holds the intermediate result 7 mentally. When this subprocedure is introduced, the show-work principle mandates that the lesson's examples write the intermediate result down. In a later lesson, the students may be taught to omit the extra writing by holding the intermediate result mentally.

---

* John Seely Brown originated repair theory (Brown & VanLehn, 1980). The present version remains true to the insights of the original version although most of the details have changed.

Austin (1962) invented felicity conditions as a way of analyzing ordinary conversations. A typical linguistic felicity condition is: In normal conversation, the speaker uses a definite noun phrase only if the speaker believes the listener can unambiguously determine the noun phrase's referent. Typically, neither the speaker nor the hearer is aware of such constraints. Yet, if a conversation violates a felicity condition, it is somehow marked, e.g., by the speaker appearing sarcastic or the hearer misunderstanding the speaker. Austin's idea has been developed by Searle (1969), Grice (1975), Gordon and Lakoff (1971), Cohen and Perrault (1979) and many others. It has become a whole new field, discourse analysis. The present work is, to my knowledge, the first application of ideas from discourse analysis to the study of learning. Two key ideas have been imported:

1. Felicity conditions are operative constraints on human behavior despite the fact that the participants are not aware of them. Textbook authors probably do not consciously realize that the lessons they write obey e.g., the show-work principle. They strive only to make the lessons effective. So too, the speakers in a conversation try only to communicate effectively, and are not aware that they obey certain felicity conditions.

2. The seeming purpose of felicity conditions is to expedite communication. In particular, there seem to be certain *inherent problems* that the listener (student) must solve. For instance, whenever a speaker uses a noun phrase, the listener must decide whether it refers to a previously mentioned object or to one that is new to the conversation. The felicity condition mentioned a moment ago helps the listener decide: If I say "the theory" right now, you will probably take it to mean the one presented in this paper. If I say "a theory," you will probably take it to mean a hitherto unmentioned theory that I will soon be telling you something about. Felicity conditions expedite communication by helping the listener solve inherent problems. Felicity conditions do not usually solve the inherent problem for the listener (student), but they do simplify the listener's task. An inherent problem for classroom knowledge communication will be discussed in a moment.

Other ideas from discourse analysis (e.g., conversational implicature — the deliberate violation of a felicity condition in order to achieve a special effect) have not yet found analogs in the domain of classroom learning. It remains an open question just how far the analogy between conversation and multi-lesson knowledge communication will go.

1.2 The methodological goal: competitive arguments for each hypothesis

The rise of AI has given psychology the tools to build computer programs that apparently simulate complex forms of cognition, such as skill acquisition, at a level of detail and precision that is orders of magnitude greater than that achieved by earlier models of cognition. Unfortunately, the potential of AI models to explain human learning (or other kinds of cognition) is largely unrealized due to methodological weaknesses. Until recently, it was rare for a model to be analyzed and explicated in terms of individual hypotheses. One was asked to accept the model *in toto*. Critics have pointed out that a typical AI/Simulation "explanation" of intelligent behavior is to substitute one black box, a complex computer program, for another, the human mind (Kaplan, 1981). Efforts at explicating programs have increased recently. Although extracting the hypotheses behind the design of the model is a necessary first step, many other issues remain to be addressed: What are the relationships between the hypotheses and the behavior? Could the given cognition be simulated if the hypotheses were violated or replaced by somewhat different ones? Would such a change produce inconsistency, or a plausible but as yet unobserved human behavior, or merely a minor perturbation in the predictions? Which alternatives if any can now be rejected in favor of the chosen hypotheses? The connection of explicit hypotheses to the data seems to me to be critical to progress in computational theories of cognition. The emphasis must be on the connection;

explication alone is only a beginning.

AI has given psychology a new way of expressing models of cognition that is much more detailed and precise than its predecessors. Unfortunately, the increased detail and precision in *stating* models has not been accompanied by correspondingly detailed and precise arguments *analyzing and supporting* them. Consequently, the new, richly detailed models of cognitive science often fail to meet accepted criteria of scientific theories. It is not new to point out that current theorizing based on computational models of cognition has been lax in providing such support (Pylyshyn, 1980; Fodor, 1981). Perhaps what is new would be to give a complex AI-based theory proper support. Not only would this be interesting in itself, but it would show that there is nothing inherent in AI-based models that prevents their use in scientifically acceptable theories.

The methodological goal of this research is to give an AI-based theory proper support. By *support*, I refer to various traditional forms of scientific reasoning such as showing that specified empirical phenomena provide positive or negative evidence regarding hypotheses, showing that an assumption is needed to maintain empirical content and falsifiability, or showing that an assumption has consequences that are contradictory or at least implausible. However, one form of support has turned out to be particularly useful. I have found that the internal structure of the theory — the way the hypotheses interact to entail empirical coverage — comes out best when the theory is compared with other theories and with alternative versions of itself. That is, a key to supporting this theory is *competitive argumentation.* In practice, most competitive arguments have a certain "king of the mountain" form. One shows that a hypothesis accounts for certain facts, and that certain variations or alternatives to the hypothesis, while not without empirical merit, are flawed in some way. That is, the argument shows that its hypothesis stands at the top of a mountain of evidence, then proceeds to knock the competitors down. Two examples of competitive arguments will be presented so that the remaining discussion of the validation problem can be conducted on a more concrete footing.

An argument for one-disjunct-per-lesson

Consider the first felicity condition listed a moment ago. A more precise statement of it is: *Learning a lesson introduces at most one new disjunct into a procedure.* In procedures, a disjunction may take many forms, e.g., a conditional branch (if-then-else). This felicity condition asserts that learners will only learn a conditional if each branch (disjunct) of the conditional is taught in a separate lesson—i.e., the then-part in one lesson, and the else-part in another.

The argument for the felicity condition hinges on an independently motivated hypothesis: mathematical procedures are learned inductively. They are generalized from examples. There is an important philosophical-logical theorem concerning induction: If a generalization (a procedure, in this case) is allowed to have arbitrarily many disjuncts, then an inductive learner can identify which generalization it is being taught only if it is given *all possible examples,* both positive and negative. This is physically impossible in most interesting domains, including this one. If inductive learning is to bear even a remote resemblance to human learning, disjunctions must be constrained. Disjunctions are one of the inherent problems of knowledge communication that were mentioned earlier.

Two classic methods of constraining disjunctions are (i) to bar disjunctions from generalizations, and (ii) to bias the learner in favor of generalizations with the fewest disjuncts. The felicity condition is a new method. It uses extra input information, the lesson boundaries, to control disjunction. Thus, there are three competing hypotheses for explaining how human learners control disjunction (along with several other hypotheses that won't be mentioned here): (i) no-disjunctions,

(ii) fewest-disjuncts, and (iii) one-disjunct-per-lesson.

Competitive argumentation involves evaluating the entailments of each of the three hypotheses. It can be shown that the first hypothesis should be rejected because it forces the theory to make absurd assumptions about the student's initial set of concepts—the primitive concepts from which procedures are built. The empirical predictions of the other two hypotheses are identical, given the lesson sequences that occur in the data, so more subtle arguments are needed to differentiate between them. Here are two:

(1) The one-disjunct-per-lesson hypothesis explains why lesson sequences have the structure that they do. If the fewest-disjuncts hypothesis were true, then it would simply be an accident that lesson boundaries fall exactly where disjuncts were being introduced. The one-disjunct-per-lesson hypothesis explains a fact (lesson structure) that the fewest-disjuncts hypothesis does not explain.

(2) The fewest-disjuncts hypothesis predicts that students would learn equally well from a "scrambled" lesson sequence. To form a scrambled lesson sequence, all the examples in an existing lesson sequence are randomly ordered then chopped up into hour-long lessons. Thus, the lesson boundaries fall at arbitrary points. (To avoid a confound, the scrambling should not let examples from late lessons come before examples from early lessons.) The fewest-disjuncts hypothesis predicts that the bugs that students acquire from a scrambled lesson sequence would be the same as the bugs they acquire from the unscrambled lesson sequence. This empirical prediction needs checking. If it is false, as I am sure it is, then the fewest-disjuncts hypothesis can be rejected on empirical as well as explanatory grounds.

This brief competitive argument sketchs the kind of individual support that each of the theory's hypotheses should be given. Such argumentation seems essential for demonstrating the psychological validity of a theory of this complexity.

However, many hypotheses of the theory are so removed from empirical predictions that it is difficult to show that they are well-motivated. This is particularly true with the hypotheses that define the representation used for the student's procedural knowledge. AI models of cognition invariably use some knowledge representation language. It is widely recognized that the architecture of the knowledge representation has subtle, pervasive effects on the model and the model's empirical accuracy. Despite this belief, most discussions of knowledge representation have been conducted on non-empirical grounds. (e.g., Can the knowledge representation cleanly express the distinction between the generic concept "elephant," the set of all elephants, and a prototypical elephant?) Knowledge representations have been treated as notational schemes, but they can be taken as theoretical assertions. It is clear that the mind contains information, and it is plausible that that information is structured. As Fodor (1975) points out, it makes sense to ask if the structure of the *mind's* information is the same as the structure of the *model's* information, where the structure of the model's information is defined by the knowledge representation language. It is just as sensical and important to ask whether hypotheses of knowledge representation are psychologically true as it is to ask whether hypotheses of learning or problem solving processes are psychologically true. However, it is considerably more difficult to ascertain the truth of hypotheses of knowledge representation since their impact on observable predictions is often quite indirect.

A major goal of the present research is to provide empirical arguments defending each principle of the theory, *including the principles that define the knowledge representation.* The arguments for the knowledge representation are intricate and depend crucially on other, more easily defended principles, such as the felicity conditions. The following section sketches one of the simplest arguments.

One-disjunct-per-lesson entails a goal stack

The argument starts with the one-disjunct-per-lesson hypothesis, brings in some data, and concludes that students' procedures employ goal stacks. A goal stack allows a procedure to be recursive. For instance, a recursive procedure for doing borrowing is:

```
Regroup (C) ≡
    1.   Add 10 to the top digit of column C.
    2.   BorrowFrom (the next column to the left of C).
BorrowFrom (C) ≡
    1.   If the top digit of column C is zero, then Regroup (C).
    2.   Decrement the top digit of column C by one.
```

It has two goals, `Regroup` and `BorrowFrom`, both taking a column as an argument. This procedure generates the following problem state sequence:

$$\text{a. }\begin{array}{r}5\;0^{1}3\\ -\;\;\;8\;6\\ \hline\end{array}\qquad \text{b. }\begin{array}{r}5^{1}0^{1}3\\ -\;\;\;8\;6\\ \hline\end{array}\qquad \text{c. }\begin{array}{r}\overset{4}{5}{}^{1}0^{1}3\\ -\;\;\;8\;6\\ \hline\end{array}\qquad \text{d. }\begin{array}{r}\overset{4}{5}{}^{1}\overset{9}{0}{}^{1}3\\ -\;\;\;8\;6\\ \hline\end{array}\ldots.$$

States *b* and *c* result from a recursive invocation of the goal Regroup. A goal stack is needed to maintain the distinction between the two invocations of `Regroup` so that, for instance, the invocation of `BorrowFrom` on the hundreds column (yielding state *c*) returns to the right invocation of `Regroup`. This recursive procedure can borrow across arbitrarily many zeros*, e.g.,

$$\text{a. }\begin{array}{r}5\;0\;0^{1}3\\ -\;\;\;\;\;8\;6\\ \hline\end{array}\quad \text{b. }\begin{array}{r}5\;0^{1}0^{1}3\\ -\;\;\;\;\;8\;6\\ \hline\end{array}\quad \text{c. }\begin{array}{r}5^{1}0^{1}0^{1}3\\ -\;\;\;\;\;8\;6\\ \hline\end{array}\quad \text{d. }\begin{array}{r}\overset{4}{5}{}^{1}0^{1}0^{1}3\\ -\;\;\;\;\;8\;6\\ \hline\end{array}\quad \text{e. }\begin{array}{r}\overset{4}{5}{}^{1}\overset{9}{0}{}^{1}0^{1}3\\ -\;\;\;\;\;8\;6\\ \hline\end{array}\quad \text{f. }\begin{array}{r}\overset{4}{5}{}^{1}\overset{9}{0}{}^{1}\overset{9}{0}{}^{1}3\\ -\;\;\;\;\;8\;6\\ \hline\end{array}\ldots.$$

The problem state sequences just given are exactly how many students borrow across zero. But this does not prove that they have a recursive borrowing procedure. They could, for instance, have a borrow procedure with two loops: one loop moves left, adding tens to zeros; the second loop moves right, decrementing as it goes:

```
Regroup (C) ≡
    1.   OriginalC ← C.
    2.   Add 10 to the top digit of column C.
    3.   C ← the next column to the left of C.
    4.   If the top digit of column C is zero, go to step 2.
    5.   Decrement the top digit of column C.
    6.   C ← the next column to the right of C.
    7.   If C≠OriginalC then go to step 5.
```

This two-loop procedure is not recursive. A goal stack is not needed to interpret it. So, two very different procedural structures are both consistent with student problem solving behavior. The one-disjunct-per-lesson hypothesis provides a way to tell which knowledge structure students have.

---

* The maximum depth that the goal stack achieves while solving a problem is proportional to the number of zeros in the problem. Since students can solve problems with arbitrarily many zeros, the goal stack has no apparent maximum depth. Evidently, this goal stack is not the same as the one(s) that are hypothesized to underlie other kinds of cognition, e.g., parsing center-embedded English sentences such as R. Stallman's pun, "The bug the mouse the cat ate bit bites."

The recursive procedure has one disjunction: the conditional statement in `BorrowFrom`. The two-loop procedure has two disjunctions: the conditional statements on lines 4 and 7. This is not an accident. Any two-loop procedure must have two disjunctions, one to terminate each loop. A recursive procedure can always get away with one disjunction. In essence, the goal stack automatically performs the second, right-moving loop as it pops.

Since only one disjunct is introduced per lesson, and the two procedures have different disjunctions, the two procedures will require different lesson sequences in order to be learned. In particular, the recursive procedure can be learned with a single lesson, assuming that the learner already knows how to borrow from non-zero digits (i.e., the student can solve 57 − 9). The lesson would have examples such as the problem state sequences given above. On the other hand, the two-loop procedure could only be learned using two lessons. The first lesson would introduce just the left-moving loop. It might use an example such as

$$\text{a.}\ \begin{array}{r} 5\;0\;0^{1}3 \\ -\quad 8\;6 \\ \hline \end{array} \qquad \text{b.}\ \begin{array}{r} 5\;0^{1}0^{1}3 \\ -\quad 8\;6 \\ \hline \end{array} \qquad \text{c.}\ \begin{array}{r} 5^{1}0^{1}0^{1}3 \\ -\quad 8\;6 \\ \hline \end{array}$$

which only does part of regrouping and stops. The second lesson would complete the teaching of the procedure by showing how to do the right-moving loop. It might use an example such as

$$\text{a.}\ \begin{array}{r} 5\;0\;0^{1}3 \\ -\quad 8\;6 \\ \hline \end{array} \quad \text{b.}\ \begin{array}{r} 5\;0^{1}0^{1}3 \\ -\quad 8\;6 \\ \hline \end{array} \quad \text{c.}\ \begin{array}{r} 5^{1}0^{1}0^{1}3 \\ -\quad 8\;6 \\ \hline \end{array} \quad \text{d.}\ \begin{array}{r} \overset{4}{5}{}^{1}0^{1}0^{1}3 \\ -\quad 8\;6 \\ \hline \end{array} \quad \text{e.}\ \begin{array}{r} \overset{4}{5}{}^{1}\overset{9}{0}{}^{1}0^{1}3 \\ -\quad 8\;6 \\ \hline \end{array} \quad \text{f.}\ \begin{array}{r} \overset{4}{5}{}^{1}\overset{9}{0}{}^{1}\overset{9}{0}{}^{1}3 \\ -\quad 8\;6 \\ \hline \end{array} \ldots$$

At this point in the argument, a difficult knowledge structure issue has been reduced to an entirely empirical question: if students have a two-loop procedure, then they must have been taught it with a lesson sequence like the two-lesson sequence above. On the other hand, if the single-lesson sequence is the only one in use, then students must have a recursive procedure. Now for the punch line: No subtraction curriculum that I have examined uses the two-lesson sequence. The curricula all use the other one. The data support the hypothesis that students have a recursive borrowing procedure, and hence, a goal stack.

An important hypothesis about knowledge representation has been supported by an entailment of the one-disjunct-per-lesson hypothesis in conjunction with a simple empirical observation. Discovering such entailments is perhaps the most important contribution that this research has to make. Most of this document is devoted to describing them. (In particular, two other arguments supporting the goal-stack hypothesis will be presented.) However, these arguments are often quite a bit more complex than the ones given above. The bug data are particularly tricky, which is why they have been avoided here. Complex inferences are a steep price (witness the length of this document!). Are they really necessary, or is there a "shallow" theory, one with more easily tested assertions, that will account for learning in this domain? I think not. The next subsection explains why.

Why not use a shallow theory?

The complexity in the theory's verification derives from the ambition that problem solving knowledge be described in enough detail to actually solve problems. To see this, consider the fate of a particular shallow theory, one of a class of stochastic learning models that explain the ubiquitous learning curves of skill acquisition (see Newell & Rosenbloom, 1981, for a review). A typical model has a pool of responses, some correct and some incorrect. The subject's response is drawn probabilistically from this pool. Learning curves are explained by simple functions which add or replace items in the pool depending on the reinforcement given the learner. Consider what it would mean to apply such a model literally to learning the skill of subtraction. A subtraction response is a sequence of writing actions. Let's say that each time a student observes the teacher's examples or answers a subtraction exercise correctly, the action sequence is added to the pool. To answer a problem, the student merely draws an action sequence from the pool and executes it. Clearly, the student's solution would have little to do with the problem, and there would never be any learning. The model makes an absurd prediction. Although one could augment the model by associating stimuli patterns with each action sequence in the pool, it's clear that there are far too many subtraction problems for this to work. One would have to postulate a matcher that finds the *closest* pool item to the problem. At last we have something that has a prayer of predicting the data. *But now all the interesting theoretical machinery is hiding in the matcher.* Many learning phenomena can be generated just by manipulating the matcher and the encoding it uses for the stimuli. The shallowness of the theory has vanished. To validate the architecture of the matcher and the representation of stimuli would require the kinds of deep inferences that this approach was supposed to avoid. The only way to get a shallow model to work that I can see is to ignore the details of the response that the subjects make, and simply classify their response as right or wrong. This gross description allowed the stochastic models to predict the appropriate learning curves with some degree of accuracy. This simplification to one bit responses, right versus wrong, characterizes much research on skill acquisition and virtually all educational research on mathematical skills.

It was just shown that a shallow theory would not work for this domain. That's unfortunate. When theories are shallow, then argumentation is easy. In a sense, the data do the arguing for you. Most experimental psychology is like this. The arguments are so direct that the only place they can be criticized is at the bottom, where the raw data is interpreted as findings. Experimental design and data analysis techniques are therefore of paramount importance. The reasoning from finding to theory is often short and impeccable. On the other hand, when theories are deep in that the derivation of predictions from remote structures is long and complex, argumentation becomes lengthy and intricate. However, the effort spent in forging them is often repaid when the arguments last longer than the theory. Indeed, each argument is almost a micro-theory. An argument's utility may often last far longer than the utility of the theory it supports. (For examples, see the discussion of crucial facts in VanLehn, Brown & Greeno, in press) As an illustration of the transition from shallow to deep theories, linguistics provides a particularly good example.

From shallow theories and deep non-theories, towards deep theories

Prior to Chomsky, syntactic theories were rather shallow and almost taxonomic in character. The central concern was to tune a grammar to cover all the sentences in a given corpus. Arguments between alternative grammars could be evaluated by determining which sentences in the corpus could be analyzed by each. When Chomsky reshaped syntax by postulating abstract remote structures, namely a base grammar and transformations, argumentation had to become much more subtle. Since transformations interacted with each other and the base grammar in complex ways, it was difficult to evaluate the empirical impact of alternative formulations of rules. Theories of syntax changed constantly and gradually as interactions are uncovered. What has been retained from the early days is not whole theories, but a loosely defined collection of crucial facts and arguments.

As Moravcsik has pointed out (Moravcsik, 1980), Chomskyan linguistics is virtually alone among the social sciences in employing deep theories. Moravcsik labels theories "deep" (without implying any depth in the normative sense) if they "refer to many layers of unobservables in their explanations.... 'Shallow' theories are those that try to stick as close to the observables as possible, [and] aim mostly at correlations between observables.... The history of the natural sciences like physics, chemistry, and biology is a clear record of the success story of 'deep' theories.... When we come to the social sciences, we encounter a strange anomaly. For while there is a lot of talk about aiming to be 'scientific,' one finds in the social sciences a widespread and unargued-for predilection for 'shallow' theories of the mind." (Moravcsik, 1980, pg. 28)

AI-based modelling efforts are certainly deep in that they postulate remote structures and mechanisms that are quite unobservable. However, very few efforts, if any, could be called proper theories. They lack arguments connecting their remote structures to empirical findings. In one sense, the history of AI-based cognitive research is the dual of linguistics' history. Throughout its history, linguistics has had a strong empirical tradition. Only lately has it adopted deep theorizing. On the other hand, AI has always had a strong tradition of deep modelling, and only recently has it begun to connect its models to observations. The present research effort is intended to be another step in that direction — putting AI-based, deep models on firm empirical footings.

1.3 Overview of the theory and the document

The preceding sections indicated the kind of skill acquisition under study, sketched a few hypotheses about it, and discussed the validation method. This section summarizes the research project by listing its main components.

(1) *Learning model.* The first component is a learning model: a large, AI-based computer program. Its input is a lesson sequence. Its output is the set of bugs that are predicted to occur among students taking the curriculum represented by the given lesson sequence. The program, named Sierra, has two main parts: (i) The *learner* learns procedures from lessons. (ii) The *solver* applies a procedure to solve test problems. The solver is a revised version of the one used to develop repair theory (Brown & VanLehn, 1980). The learner is similar to other AI programs that learn procedures inductively. For instance, ALEX (Neves, 1981) learns procedures for solving algebraic equations given examples similar to ones appearing in algebra textbooks. LEX (Mitchell et. al., 1983) starts with a trial-and-error procedure for solving integrals and evolves a more efficient procedure as it solves practice exercises. Sierra's learner is similar to LEX and ALEX in some ways (e.g., it uses disjunction-free induction). It differs in other ways (e.g., it uses lesson boundaries crucially, while the instruction input to ALEX and LEX is a homogeneous sequence of examples and exercises). In particular, Sierra is the first AI learner to use rate constraints (described in the next chapter). As a piece of AI, Sierra's learner is a modest contribution. Of course, the goal of this research is not to formulate new ways that AI programs can learn.

(2) *Data from human learning.* The data used to test the theory come from several sources: the Buggy studies of 2463 students learning to subtract multidigit numbers (Brown & Burton, 1978; VanLehn, 1982), a study of 500 students learning to add fractions (Tatsuoka & Baille, 1983), and various studies of algebra errors (Greeno, 1982; Wenger, 1983). The data from subtraction play the most prominent role since they derive from the largest sample and the most objective analysis methods. Bugs from the other procedural skills play the secondary, but still important role of testing the across-task generality of the theory. As of this writing, only the subtraction data have been analyzed. A formal assessment of the theory's task-generality must be delayed for another report.

(3) *A comparison of the model's predictions to the data.* The major empirical criterion for the theory is *observational adequacy:* (i) the model should generate all the correct *and buggy* procedures that human learners exhibit, and (ii) the model should not generate procedures that learners do not acquire, i.e., star bugs. Although observational adequacy is a standard criterion for generative theories of natural language syntax, this is the first AI learning theory to use it.

(4) *A set of hypotheses.* As mentioned above, early AI-based theories of cognition used only the three components listed so far: a model, some data, and a comparison of some kind. Such an "explanation" of intelligent human behavior amounts to substituting one black box, a complex computer program, for another, the human mind. Recent work in automatic programming and program verification suggests a better way to use programs in cognitive theories: The theorist develops a set of specifications for the model's performance. These serve as the theory's hypotheses about the cognition being modelled. The model becomes a tool for calculating the predictions made by the combined hypotheses. The present theory has 32 such hypotheses. The felicity conditions listed earlier are three of the 32. The goal stack hypothesis is another.

(5) *A demonstration that the model generates all and only the predictions allowed by the hypotheses.* Such a demonstration is necessary to insure that the success or failure of the model's predictions can be blamed on the theory's hypotheses and not on the model's implementation. Ideally, I would prove, line-by-line, that the model satisfies the hypotheses. This just isn't practical for a program as complex as Sierra. However, what has been done is to design Sierra for transparency instead of efficiency. For instance, Sierra uses several generate-and-test loops where the tests are hypotheses of the theory. This is much less efficient than building the hypotheses into the generator.* But it lends credence to the claim that the model generates exactly the predictions allowed by the hypotheses.

(6) *A set of arguments, one for each hypothesis, that shows why the hypothesis should be in the theory, and what would happen if it were replaced by a competing hypothesis.* This involves showing how each hypothesis, in the context of its interactions with the others, increases observational adequacy, or reduces degrees of freedom, or improves the adequacy of the theory in some other way. The objective is to analyze *why these particular* hypotheses produce an empirically successful theory. This comes out best in competitive argumentation. Each of the 32 hypotheses of the theory has survived a competitive argument.

The structure of the remainder of this document

The next chapter presents the model (component 1 in the list above) and discusses its observational adequacy (component 3). The remaining chapters present the hypotheses of the theory (component 4) and the arguments supporting them (component 6). They are grouped into three levels:

1. The *architecture level* establishes the basic relations between lesson sequences and the acquired skill. The acquired skill is sometimes called a *core* procedure because it cannot be directly observed. The architecture level also establishes the basic relations between the core procedure and observable behavior during problem solving. Such behavior is sometimes called the *surface* procedure despite the fact that it is occasionally rather non-procedural in character. The felicity conditions and the hypotheses defining local problem solving are defined in the architecture level. These hypotheses are expressed without using a formal representation for core procedures. This allows the architecture level's hypotheses to be defended at a relatively high level of detail using broad, general observations about the character of learning and problem solving in this domain.

---

* It takes Sierra about 150 hours of Dorado time to process a single subtraction lesson sequence. However, Sierra is a multiprocessor program that can be run unattended at night using as many Dorados as it can find on our local Ethernet. It sometimes takes only a few days to process a lesson sequence. This style of research would be infeasible without networks of fast Lisp machines, such as the Dorado.

2. The *representation level* defines a formal representation for core procedures. The representation level takes the architecture level as given, then uses the data to settle representational issues. From another viewpoint, the hypotheses of the representation act as absolute constraints on learning and problem solving (as opposed to relative constraints). Given a particular lesson sequence, these absolute constraints determine a large space of possible core procedures that could be acquired from it.
3. The *bias level* establishes relative constraints on learning and problem solving. Whereas the representation level defines a space of possible core procedures, the bias level defines an *ordering relation* over the core procedures in the space and states that learners choose core procedures that are maximal in this ordering. The bias level takes both of the higher levels as given.

The three levels can be introduced by drawing analogies to several prominent traditions in cognitive science. The architecture level is like a Piagetian theory in its broad-brush treatment of cognition. The representational level is like a Chomskyan theory of syntax in that it is concerned with the structure of mentally held information. The bias level is like Newell and Simon's theory of human problem solving in its attention to detailed individual behavior and its use of computer simulations. Each level has its own objectives, and each uses the data in different ways.

These three levels contain mostly competitive argumentation, and their format refects this. Each chapter argues a single issue. The chapter begins by laying out the competing hypotheses. It indicates which hypothesis is ultimately chosen for inclusion in the theory. The body of the chapter shows why the other hypotheses lead to a less adequate theory. The chapter ends with a summary of the arguments and a formal statement of the adopted hypothesis. Chapter introductions and conclusions have been written so that they can be understood without reading the chapter's body. As a further aid to browsing readers, each level has a summary chapter that synopsizes the arguments and hypotheses discussed in the level. These summaries may be read without having read the level itself.

In addition to producing a six-component theory, the research produced a few surprises. Mentioning one of them is perhaps a fitting end for this introductory chapter.

Felicity conditions > teleological rationalizations

From the outset of this research, it was clear that learning depended strongly on the examples used in instruction. It was also clear that learning could not depend *solely* on the examples. Some other kind of information had to be involved. The issue was, what information was being provided by the curriculum, and what information did the student already have? A highly plausible hypothesis was that learners possessed *teleological rationalizations* as prior knowledge. Teleological rationalizations express the learner's presupposition that procedures have purposes and hence that the "right" generalization of the examples to make is the one that leads to a procedure with recognizable purposes for each of its parts. So, the learner acquires only subprocedures whose content can be rationalized vis à vis the learner's general notions of purposes for procedures. For instance, a simple rationalization is one that views a new step (subprocedure) as preparation for an already known step (Goldstein's "setup step" schema, 1974).

This view was comfortably in line with the common view that *a procedure can be learned only to the extent that it is meaningful to the learner.* Here, teleological rationalizations expressed the meaning that learners give procedures. The rationalizations may not impart the correct semantics (the semantics the teacher intended), so the procedure acquired may not be a correct one. Yet they do give the procedure some kind of semantics.

The view that learning is necessarily meaningful seems now to be false for the present domain. I was unable to detect any widely held teleological rationalizations. Moreover, those that I guessed might be held, perhaps scattered idiosyncratically in the population, did not constrain acquisition enough to explain the data. On the other hand, certain felicity conditions were discovered that were strong enough to eliminate many of the ambiguities that teleological rationalizations were supposed to settle. Although I had guessed a few felicity conditions some years ago, I was surprised to discover the show-work principle, and even more surprised to see how much constraint the felicity conditions placed on learning. Not only do the felicity conditions do as much or more work than teleological rationalizations, they appear to be held by *all* individuals, while the set of teleological rationalizations would have to be subject to individual differences. To top it off, the new felicity conditions are much simpler than teleological rationalizations. For several reasons, therefore, teleological rationalizations have been excluded from the theory. It currently seems that rationalization of subprocedures might be more in the mind of the observer (me) than in the student's mind.

Omitting teleological rationalizations in favor of felicity conditions changes the overall character of the learning theory. Teleological rationalizations could give the acquired procedure a meaning, albeit a potentially incorrect meaning, by relating it to general teleological knowledge about procedures. The felicity conditions and the constraints on representation essentially allow the procedure to be built from primitives in apparent isolation from other knowledge. This result is consonant with the widely held impression that mathematical procedures are often understood syntactically (Resnick, 1982). It tends to refute the also common view that procedures can only be learned if they have some meaning for the learner.

Chapter 2
Sierra, the Model

This chapter concerns the model, a computer program named Sierra. The term "model" is used in a narrow way to mean an artifact whose structure and performance is similar, in certain ways, to the cognition under study. Under this usage, "model" is not synonymous with "theory." The model is a thing; the cognition is a thing; the theory asserts how the two things relate. In physics, a model is usually a system of equations, which the theory relates to the physical system being studied. A physical theory might say, for instance, which variables in the model are measurable, which equations represent natural laws, and which equations represent boundary conditions that are idiosyncratic to particular experiments. Of course, theories include much more than just assertions. This theory includes, for instance, a tacit set of distinctions or ways of analyzing the cognition. It includes an analysis of how the data and the model's performance relate. It includes, of course, the hypotheses and the competitive arguments that support them. Indeed, everything in this document is included in the theory. This chapter, however, merely describes the model.

AI-based models are plagued with a methodological problem that occurs in mathematical models as well, although it is less severe there. A typical mathematical model has parameters whose values are chosen by the experimenter in such a way that the model's predictions fit the data as closely as possible. Certain parameters, often called *task* parameters, encode features of the experimental task (e.g., what kind of stimulus material is used). Other parameters, called *subject* parameters, encode aspects of individual subjects' cognition or performance. There are other kinds of parameters as well. The difference between the parameters lies in how they are used in fitting the model's predictions to the data. Subject parameters may be given a different value for each subject. Task parameters get a different value for each experimental task, but that value is not permitted to vary across individual subjects. When AI-based models have been used for cognitive simulations, there has often been considerable obscurity in the boundary between what is meant to be true of all subjects, and what is meant to be true of a particular subject. Often, the same knowledge base (rule set or whatever) is used for both subject parameters and task parameters. Yet it is critical that theories, even if they use non-numeric parameters, identify which of the model's components and principles are universal, which are task specific, and which may be tailored to the individual. But this is just the beginning of the problem. Even if the kind of tailoring has been clearly delineated as universal, task, subject, or whatever, there remains a difficult issue of determining how much influence the theorist can exert over the model's predictions by manipulating the parameters' values. In a mathematical model, such power is often measured by counting degrees of freedom or performing a sensitivity analysis. For models whose "parameters" are knowledge bases or rule sets, there is, as yet, no equivalent measure of tailorability. It is crucial, however, that the tailorability of such models be better understood. A model whose fit to the data depends on the cleverness of the theorist writing the rules doesn't really tell us much of interest. Understanding Sierra's tailorability *and reducing it* have been major concerns in developing this theory. Reduced tailorability is as much a goal for the theory as observational adequacy. Many of the hypotheses that are presented in later chapters are adopted just because they reduce the tailorability of the model.

In addition to describing the model, this chapter discusses its observational adequacy and tailorability in the context of one particular experiment, called the Southbay experiment, wherein 1147 subtraction students were tested. As Sierra is described, its various parameters will be illustrated by mentioning the values that they are given in tailoring the predictions to fit the

Southbay data. The effects of varying these values will also be discussed as a rough sensitivity analysis of the model. The last section of the chapter assesses the observational adequacy of the model with respect to the Southbay data. This section is the only place in the document where observational adequacy will actually be measured, and for good reason. Having produced the numbers, it will be argued that there are no magic thresholds for such measurements. One can't tell from the numbers whether the theory is good or bad. Measuring observational adequacy is only useful for comparing the theory to other theories, and in particular, for comparing it to other versions of itself. That is, observational adequacy is useful primarily as an empirical criterion for competitive argumentation. Although it is interesting to go through a full-fledged measurement of observational adequacy, once is enough. Thereafter, observational adequacy will be used only as part of competitive argumentation.

This chapter presents the model, Sierra, in enough detail that it can be duplicated. At one point, this document had a separate chapter for this purpose. However, it became so redundant with this chapter that the two were merged. Meeting this objective sometimes involves presenting technical details that are theoretically irrelevant, but necessary for understanding how Sierra works. The reader may skim over these details. For reading the remaining chapters, it suffices to grasp just the broad outline of Sierra. In particular, only sections 2.1 and 2.2 are really necessary; the others can be skipped on a first reading. What this chapter does *not* do is to justify the model. Motivating, justifying and explaining why the model is the way it is — these are proper functions for competitive argumentation. Competitive argumentation is the province of the remaining chapters.

2.1 The top level of Sierra

Sierra generates the theory's predictions about a certain class of experiments. In order to understand the way Sierra makes predictions, it helps to first understand the experiments. The experiments use the following procedure. For each school district, the experimenter ascertains what textbooks are used in teaching the given skill and when it is scheduled to be taught. In the case of the Southbay experiment, subtraction was taught from the middle of the second grade to the end of the fourth grade. Classrooms and testing dates are selected so as to sample this time span fairly evenly. Next, the experimenter meets with the participating teachers in order to brief them and to give them blank test forms, such as the one in figure 2-1. Soon thereafter, the teachers hand out the test sheets to their students, who work them alone with no time limit. The teacher collects the test sheets and mails them to the experimenter for analysis. An important point to notice is the temporal relationship between the administration of the test and the episodes of lesson-learning. Suppose that a certain curriculum has ten lesson, call them L_1, L_2, L_3, ... L_{10}. Some of the students have taken only lesson L_1 at the time they are tested, while other students have taken only L_1 and L_2, and so forth. Although a few students have taken the whole lesson sequence at the time they were tested, many data come from students who have traversed only a prefix of the lesson sequence.

This motivates the top-level design of Sierra, which is sketched in figure 2-2. Sierra's major components are called the *learner* and the *solver.* Sierra's learner is given a lesson, L_1 and an initial knowledge state, KS_0. (Actually, it is given formal representations of L_1 and KS_0. The formal representations will be discussed later.) The learner produces a new knowledge state, KS_1. It may produce more than one knowledge state, but just one is shown in the diagram for simplicity's sake. In order to generate predictions about students who have only taken L_1, KS_1 is given to Sierra's solver along with (a formal representation of) a diagnostic test, T. The solver produces a set of solved tests, ST_1. Each solved test in ST_1 represents a testable prediction about student behavior.

Subtraction Test

Name____ Grade____________

Teacher______________________________________ Date____________

647 - 45	885 - 205	83 - 44	8305 - 3
50 - 23	562 - 3	742 - 136	106 - 70
716 - 598	1564 - 887	6591 - 2697	311 - 214
1813 - 215	102 - 39	9007 - 6880	4015 - 607
702 - 108	2006 - 42	10012 - 214	8001 - 43

Figure 2–1
One of the test forms used to collect the subtraction data.

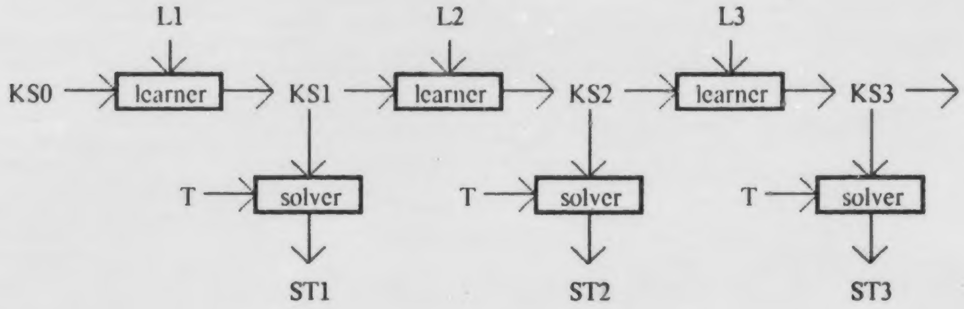


Figure 2–2
The top level of Sierra

Figure 2–2 shows that KS_1 is also given to Sierra's learner along with lesson L_2. The learner produces KS_2, which corresponds to the knowledge state of students who have seen the first two lessons of the sequence before being tested. KS_2 is passed to the solver and processed in the same way that KS_1 was. This produces predictions about the performances of students who have taken the first two lessons. Similarly, predictions are produced for students at all other stages of training, including students who have completed the lesson sequence.

The model's predictions are the sets of solved tests, the ST_i. In principle, they could be compared directly to observed test solutions, the ones mailed in by the teachers. For several mundane reasons, this not practical. Several test forms are used in the schools in order to thwart students who look at their neighbor's paper. If the ST_i were to be compared directly to the observed test solutions, Sierra would have to be run many times, each with a different test form as T. Also, direct comparison of test solutions would have to deal with the slips that students make. A single facts error (e.g., $7-5=3$) would prevent an observed test solution from matching a predicted test solution. Some model of slip-based "noise" would have to be applied in the matching process. Even if such a slip model were quite rudimentary, it would have to be carefully and objectively parameterized lest it cause Sierra to be unfairly evaluated. Debuggy is used to solve these problems. Debuggy is equipped to deal with multiple test forms and with slip-based noise (see Burton, 1981). Its slip model, which was developed long before this theory, has been carefully honed in the process of analyzing thousands of students' work. Debuggy is used to analyze both predicted and observed test solutions. When Debuggy analyzes a solved test, it redescribes the test solution as a set of bugs. Sometimes the set is a singleton, but often a test solution, even one generated by the model, requires several bugs to accurately describe its answers. Given these bug sets, matching is simple. A predicted test solution matches an observed test solution if Debuggy converts both to the same set of bugs.

This way of comparing test solutions has an added benefit. It affords a natural definition of partial matching: two test solutions partially match if the intersection of their bug sets is non-empty. Partial matching is a useful investigative tool. For instance, if the model generates a test solution whose bug set is {A B}, and there is a test solution in the data whose bug set is {A B C}, then partial matching allows one to discover that the model is accounting for most of the student's behavior, but the student has a bug C that the model does not generate. If the two solved tests were compared directly, they would not match at all (say), yielding the experimenter no clue as to what is wrong. So comparing solved tests via Debuggy not only handles multiple test forms and noise, it promotes a deeper understanding of the empirical qualities of the model.

Sierra has a natural internal chronology. KS_2 is necessarily produced after KS_1. Perhaps this chronology makes true temporal predictions. In the Southbay experiment, for instance, the testing dates and the textbooks are known, so the approximate locations of each student in the lesson sequence can be inferred. It would be remarkable if an ST_i matched only the test solutions of students between the lessons corresponding to L_i and L_{i+1}. Longitudinal data could even be predicted, provided that the model is changed slightly*. Given that a student's test solution matched a solved test in ST_i, one could predict that a later test solution would have matched some test in ST_j for $j \geq i$. In fact, one may be able to predict that the second test would have to match certain tests of the ST_j, because only those test solutions are derived from the knowledge state KS_i that the student seemed to have at the time of the first test. Although Sierra was not designed for it, Sierra can make predictions about the chronology of skill acquisition.

Even a cursory examination of the data reveals that such chronological predictions would turn out rather poorly. Partly, this is because the experiments didn't carefully assess chronological factors. Although the general locations of students in the curricula were recorded, there is no way to know an individual's case history in any detail. In the Southbay experiment, for instance, some young students who had only taken the first few subtraction lessons could already subtract perfectly. Perhaps they learned at home or with special tutoring from the teacher. Keeping careful track of how much instruction students actually receive is, of course, a major problem in any longitudinal study. That is why I have concentrated on an a-chronological account of skill acquisition.

Even if excellent longitudinal data were available, I doubt that Sierra's prediction of them would be anywhere near the mark. Basically, this theory attacks only half of school-house learning: knowledge communication. Knowledge compilation is the other half. It deals with tuning, restructuring and other changes in the memory trace that occur with practice. Knowledge compilation undoubtedly affects the chronology of skill acquisition. Since Sierra doesn't model practice effects, it would be wild to take its chronology seriously as a reflection of the chronology of human learning.

The model's empirical quality is measured in an a-chronological way. All the ST_i are simply unioned. This creates a large set of predicted solved tests — call it PST. Similarly, the observed solved tests are collected together into a large set, call it OST, without regards to when the students were tested. The solved tests in both PST and OST are redescribed as bug sets using Debuggy. Empirical quality is measured by their overlap:

$OST \cap PST$	is the set of *confirmed* predictions. It should be large.
$OST - PST$	is the set of observed behaviors that the model doesn't account for. It should be small.
$PST - OST$	is the set of predictions that are not confirmed by the data. Some of these predictions will be absurd: star bugs. There should be very few of these. The rest are *outstanding* predictions. Further data may verify them. It doesn't matter how large the set of outstanding predictions is, as long as its members are all plausible predictions.

---

* Diagnostic testing undoubtedly has some effect on a student's knowledge state. If the model were used to make predictions about students who are tested twice, it would be advisable to route the solver-modified KS_i back up to the learner. This is not done in the current version of Sierra because almost all of the data come from students who were tested just once. Some were tested twice, but without intervening instruction.

This overlap-based measure is traditionally called observational adequacy. It is the only empirical measure that is used in validating the present theory.

This section describes the way that the main parts of the model — the learner, the solver, the KS_i, the L_i and the ST_i — hook together. It also describes the way that the theory's observational adequacy is assessed. With these frameworks in hand, it's time to plunge into a detailed description of the model. The first section describes the formal representations for lessons and solved tests, the L_i and the ST_i. The next few sections describe the knowledge representation, the KS_i. Then the learner is described, with a slight pause of some general remarks about induction. The solver is described next, but somewhat sketchily since it is substantially the same as the one described in Brown & VanLehn (1980). The last section reveals the observational adequacy of the theory, vis-à-vis the Southbay experiment.

2.2 The representation of observables: lessons and diagnostic tests

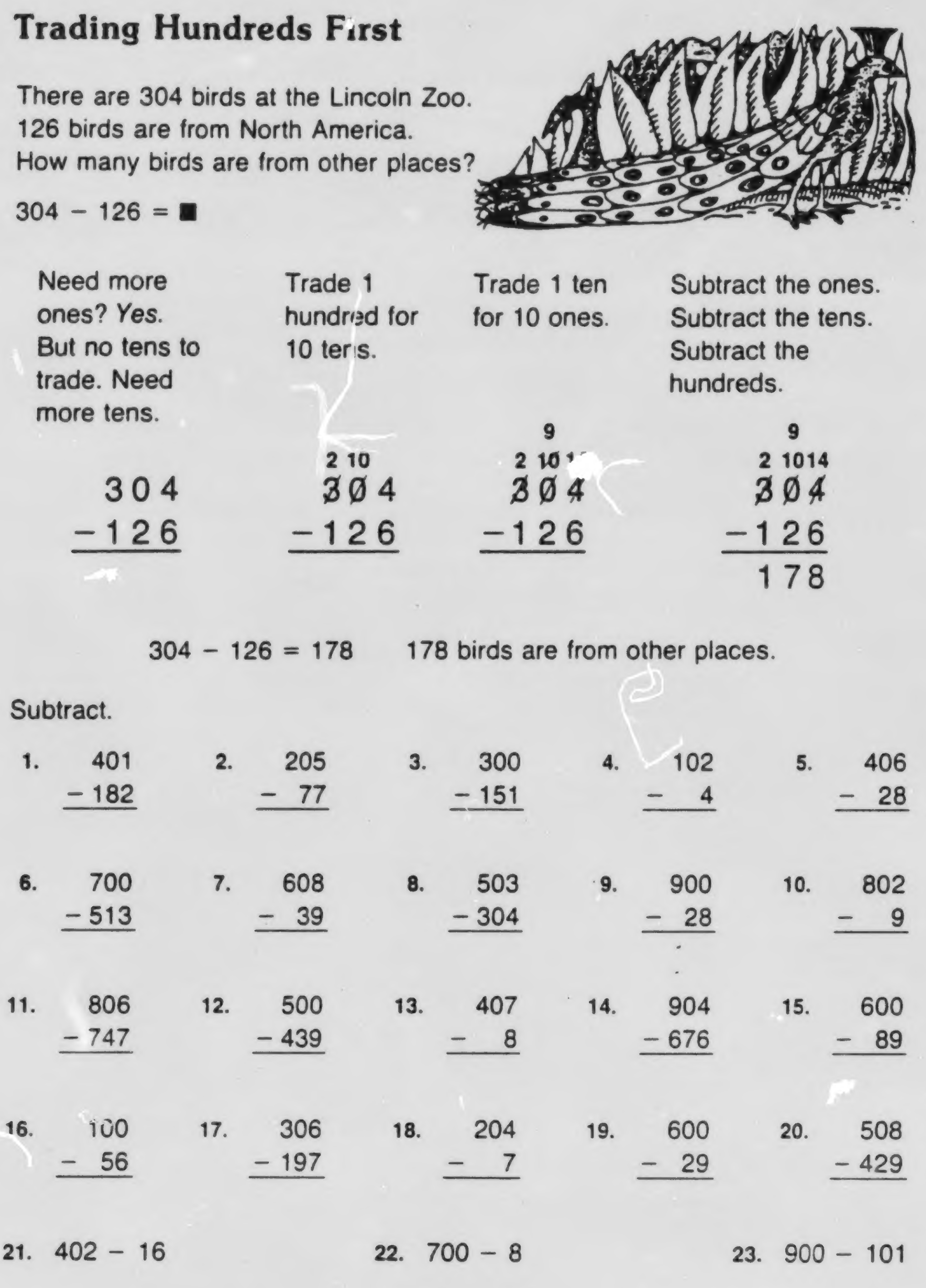
As mentioned, Sierra takes three inputs: (1) a lesson sequence, L_i, (2) a diagnostic test, T, and (3) a student's initial knowledge state, KS_0. Sierra's output is a large set of solved tests, the ST_i. Although the theorist must guess what the initial student knowledge state is, the other inputs and outputs represent observable quantities. Sierra's accuracy as a model depends somewhat on how these observable quantities are formalized. This section discusses the representations used for the observables: lessons, tests, and solved tests. The formal definitions are tediously simple:

A *lesson sequence* is a list of lessons.
A *lesson* is a pair: it is a list of examples followed by a list of exercises.
An *example* is a sequence of problem states.
An *exercise* is a single problem state.
A *test* is a list of exercises.
A *solved test* is a list of examples.
A *problem state* is a set of symbol-position pairs, where a symbol's position is represented by the Cartesian coordinates of the symbol's lower left corner and its upper right corner.

These definitions all depend on the representation of problem states, so it is worth a moment to examine that definition in detail. Problem state *a* (see below) represents *b*, and *c* represents *d*.

```
a.    ((HBAR (12 17 20 17))     b.      5 0 7      c.    ((5 (12 10 14 12))
       (- (12 17 14 19))              -   2 9             (x (14 10 16 12))
       (5 (14 19 16 21))              -------             (+ (16 10 18 12))
       (0 (16 19 18 21))                                  (1 (18 10 20 12)))
       (7 (18 19 20 21))
       (2 (16 17 18 19))        d.      5x+1
       (9 (18 17 20 19)))
```

The formal representations, *a* and *c*, are sets of pairs. Each pair represents an instance of a symbol at a place. The first element of the pair is the symbol, usually an alphanumeric character or a special symbol like `HBAR`, which stands for a horizontal bar. The second element of the pair is a tuple of four Cartesian coordinates that represent the symbol's position. The details of representing the symbol's position don't matter. The point is only that a problem state is little more than a picture of a piece of paper or a chalkboard. It is not an interpretation or parsing of the symbols. For instance, the problem state does not force the model to treat 507−29 as two rows, or as three columns, or as rows and columns at all. How the problem state is parsed is determined by a component of the student knowledge state, called the grammar. Grammars are described in a later section.



Trading Hundreds First

There are 304 birds at the Lincoln Zoo.
126 birds are from North America.
How many birds are from other places?

304 − 126 = ■

Need more ones? *Yes.* But no tens to trade. Need more tens.	Trade 1 hundred for 10 tens.	Trade 1 ten for 10 ones.	Subtract the ones. Subtract the tens. Subtract the hundreds.
$\begin{array}{r}304\\-126\\\hline\end{array}$	$\begin{array}{r}2\ 10\ \ \\ \cancel{3}\cancel{0}4\\-126\\\hline\end{array}$	$\begin{array}{r}9\ \ \\ 2\ 10\ 14\\ \cancel{3}\cancel{0}\cancel{4}\\-126\\\hline\end{array}$	$\begin{array}{r}9\ \ \\ 2\ 10\ 14\\ \cancel{3}\cancel{0}\cancel{4}\\-126\\\hline 178\end{array}$

304 − 126 = 178   178 birds are from other places.

Subtract.

1. 401 − 182	2. 205 − 77	3. 300 − 151	4. 102 − 4	5. 406 − 28
6. 700 − 513	7. 608 − 39	8. 503 − 304	9. 900 − 28	10. 802 − 9
11. 806 − 747	12. 500 − 439	13. 407 − 8	14. 904 − 676	15. 600 − 89
16. 100 − 56	17. 306 − 197	18. 204 − 7	19. 600 − 29	20. 508 − 429

21. 402 − 16   22. 700 − 8   23. 900 − 101

Figure 2-3
A page from a third grade mathematics textbook.
(Bitter, G.G., Greenes, C.E., Sobel, M.A., Hill, S.A., Maletsky, E.M., Shufelt, G., Schulman, L. & Kaplan, J. *McGraw-Hill Mathematics*, New York: McGraw-Hill, 1981. Reproduced with permission.)

How faithful are these formal representations to real curricula and real diagnostic tests? Faithfulness of tests is easy to obtain. Earlier, in figure 2–1, a copy of one of the diagnostic tests was presented. It can be quite faithfully represented as a sequence of exercises (problem states). Accurately representing a lesson is not so simple. Figure 2–3 is a black-and-white rendering of a page from a third grade textbook. It is the first page of a two-page lesson that introduces borrowing across zero. The lesson leads off by posing a word problem. It is followed by an example, 304 – 126. The example consists of four problem states. (In the textbook, the four problem states are differentiated by four lightly colored boxes, which are not reproduced here.) The rest of the page contains exercises. However, the teacher undoubtedly works the first few exercises on the chalkboard. In effect, this converts the first few exercises into examples. The second page of the lesson contains more exercises and a few word problems.

Sierra's lessons differ from real lessons in several ways. In keeping with the hypothesis that knowledge communication, in this domain, is inductive, Sierra's examples lack the English commentary that the real examples have. Its lessons also omit word problems, pictures and analogies with concrete objects like coins or blocks. They have only examples and exercises. Figure 2–4 summarizes the formal lesson corresponding to the real lesson of figure 2–3. Figure 2–4a shows the problem state sequence that represents the first example. On the assumptions that the teacher would work this example on the board, the intermediate problem states that are not pictured in the textbook are shown in the formal version of the example. Figure 2–4b summarizes the whole lesson. The formal lesson is considerably shorter than the real lesson: it has fewer examples (probably) and many fewer exercises. Since Sierra is slow, I have kept the lessons as short as possible. This makes it more difficult to keep the lessons faithful to the real lessons. In a set of examples and exercises, there might be idiosyncratic features that happens to be held by all of them. The difficulty is that the formal lesson might have different idiosyncracies than the real lesson. Since Sierra's learner is mildly sensitive to such idiosyncracies, this difference can't be ignored. So insuring the faithfulness of lessons is not trivial.

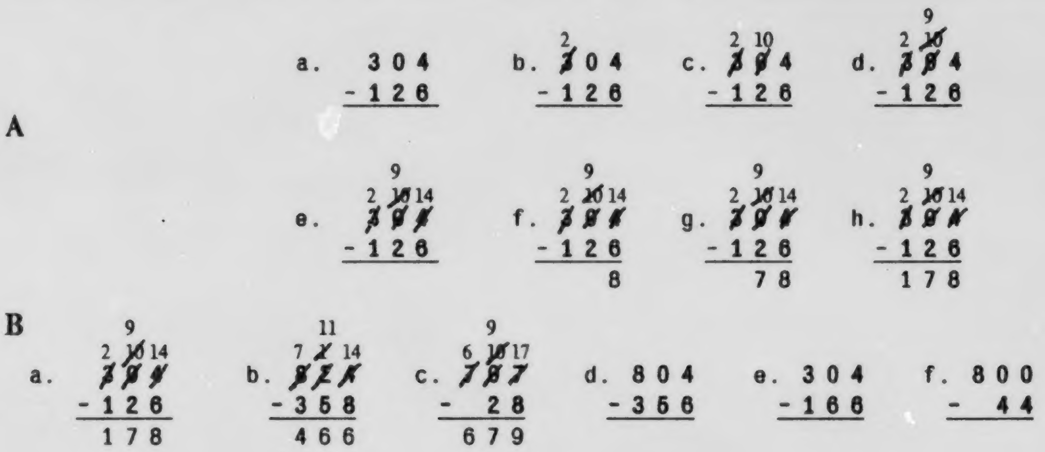


Figure 2–4
A shows the first example of the lesson as a problem state sequence (omitting crossing-out actions).
B summarizes the three examples and three exercises that constitute the formal lesson.

How an individual lesson is represented was just discussed. A curriculum is formalized as a sequence of lessons. Some of the tacit issues behind formalizing curricula are best discussed in the context of specific cases. Two textbooks were used by the schools that participated in the Southbay experiment: the 1975 edition of Scott-Foresman's *Mathematics Around Us*, and the 1975 edition of Heath's *Heath Elementary Mathematics*. From these textbooks, three formal lesson sequences were eventually derived. (This development is interesting partly because it is a clear case of tailoring a parameter of the model.) Some curricular features that at first seemed to be important turned out not to be. In particular, both textbooks introduce multicolumn subtraction using special notational devices that emphasize the columns and their names. Scott-Foresman labels the digits, as in *a* below, then switches to column labels, as in *b*, then finally to standard notation, as in *c*.

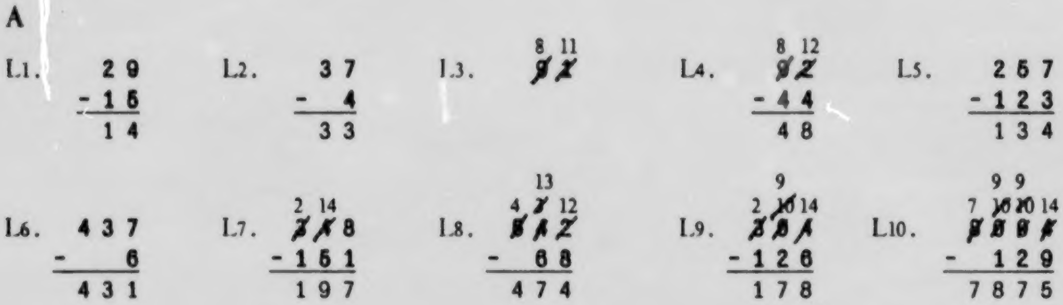
```
a                          b   tens | units           c
   3 tens   7 ones                  |                       3 7
 -          5 ones                3 |   7                 -   2
 _________________          +       |   5                 _____
                                ____|____
 __tens   __ones                  3 |   1
```

Heath starts with *b*, then switches to *c*. Generally, the textbooks would stick with their first notation until the second lesson on borrowing. Then they would shift to the next notation, and teach the last few lessons over again using the new notation. Sierra's first formal lesson sequences copied these notational excursions faithfully — lines, words and all. It was found that these extra markings made no significant difference in Sierra's predictions. When the extra markings were omitted from the examples, the resulting core procedures generated the same bugs. This finding suggests that the extra markings are included in the examples because they help students learn a grammar for subtraction notation. Sierra is given a grammar instead of learning it (this is discussed in the next section), so it receives no benefit from the extra markings. The lesson sequences that were ultimately arrived at use only the standard notation (type *c* above). This makes them shorter, saving Sierra time.

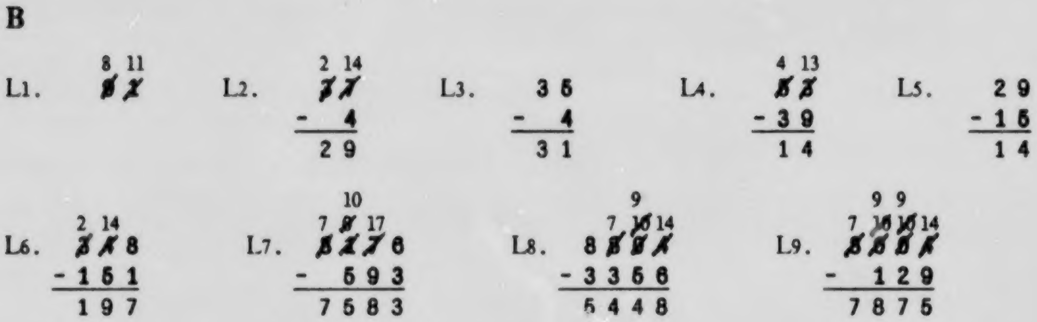
There are a few more minor differences between the real lesson sequences and the formal ones that will be discussed later. A major difference, perhaps the most important difference, will be discussed next. Figure 2-5 shows the lesson sequences for Heath (H) and for Scott-Foresman (SF). Note that both H and SF involve a special lesson on regrouping. (In the McGraw Hill lesson of figure 2-3, this subskill is called "trading" instead of "regrouping.") The regrouping lesson is L_3 in H and L_1 in SF. The regrouping lesson does not teach how to answer subtraction problems per se. It teaches how to do a subprocedure, regrouping, that is later incorporated into the subtraction procedure. It is possible that students may not understand that this regrouping lesson has anything to do with subtraction. After all, students are being taught many other skills (e.g., addition) as they are taught subtraction, yet few develop subtraction bugs by mistakenly incorporating lessons from addition or other skills. Very little is known about how students filter irrelevant lessons out of a skill's lesson sequence. But whatever this filter is, students may use it to filter out the regrouping lesson as well as addition lessons. To test this, a third lesson sequence was constructed by deleting the regrouping lesson from H. This lesson sequence, HB, turned out to be quite productive. It generated eight observed bugs that would not otherwise have been generated*. So it seems that some students take regrouping to be a part of subtraction and some don't. Lesson sequence HB is included with the other two in generating the Southbay predictions.

---

* The observed bugs generated by HB alone are: Borrow-Don't-Decrement-Zero-Unless-Bottom-Smaller, Borrow-Across-Second-Zero, Borrow-From-One-Is-Nine, Borrow-From-One-Is-Ten, Borrow-From-Zero, Borrow-From-Zero-Is-Ten, Stops-Borrow-At-Multiple-Zero, Forget-Borrow-Over-Blanks, and Smaller-From-Larger-Instead-of-Borrow-Unless-Bottom-Smaller.


A
L1. 29 - 15 14
L2. 37 - 4 33
L3. 8 11 92
L4. 8 12 92 - 44 48
L5. 257 - 123 134
L6. 437 - 6 431
L7. 2 14 348 - 151 197
L8. 13 4 3 12 542 - 68 474
L9. 9 2 10 14 304 - 126 178
L10. 9 9 7 10 10 14 8004 - 129 7875

L_1. Solving two columns.
L_2. Handling partial columns.
L_3. Regrouping.
L_4. Simple borrowing.
L_5. Solving three columns without borrowing.
L_6. Handling non-final partial columns.
L_7. One borrow in three columns.
L_8. Two adjacent borrows (3 columns).
L_9. Borrowing from zero (3 columns).
L_{10}. Borrowing from multiple zeros.


B
L1. 8 11 91
L2. 2 14 33 - 4 29
L3. 35 - 4 31
L4. 4 13 53 - 39 14
L5. 29 - 15 14
L6. 2 14 348 - 151 197
L7. 10 7 8 17 8176 - 593 7583
L8. 9 7 10 14 8804 - 3356 5448
L9. 9 9 7 10 10 14 8004 - 129 7875

L_1. Regrouping.
L_2. Borrowing in 3-digit problem.
L_3. Non-borrowing in 3-digit problem.
L_4. Borrowing in 4-digit problem.
L_5. Non-borrowing in 4-digit problem.
L_6. Solving 3-columns, with one borrow.
L_7. Adjacent borrows, in 4-column problem.
L_8. Borrowing from zero (4 columns).
L_9. Borrowing from multiple zero.

Figure 2–5
(A) The H lesson sequence. (B) The SF lesson sequence.
Sample problems are shown above, topics are listed below.

Of the three inputs to the model — the initial knowledge state KS_0, the test T, and the lesson sequence L_i — the one that has the most effect on the model's predictions is the lesson sequence. In fact, for the Southbay experiment, only three runs of Sierra were used, one for each of H, SF and HB. The same KS_0 and T were used with each run because they have very little effect on the ultimate output.

2.3 The representation of procedures

This section discusses the representation of student knowledge. The particular representation to be described is the ninth in a series of representations, which began with a homebrew version of the OPS2 production system language (Forgy & McDermott, 1978). The present knowledge representation is very much the product of empirical honing. It is not a mere notation. It expresses empirical hypotheses of the theory. This methodological stance deserves comment.

Every AI-based models of cognition that I know of has some kind of knowledge representation language. Various kinds of production systems are common, for example. Often, one reads that a theorist has revised a widely available language in order to make it "better" for the model under development, yet no theoretical claims are attached to this implicit assertion of optimality. The knowledge representation language is being treated as a mere notation that the theorist may change at will in order to make it more convenient to use.

However, one often sees conjectures that the knowledge representation is more than a mere notation (e.g., "We confess to a strong premonition that the actual organization of human programs closely resembles the production system organization." Newell & Simon, 1972, pg. 803). Fodor (1975) argues that such conjectures may be legitimate as scientific hypotheses about the mind. It is clear that the mind holds information (knowledge) and it is plausible that this information is structured in some way. Therefore, it makes sense to ask what that structure is. One way to find out what the structure of knowledge is (in Fodor's terms, to determine the mind's *mentalese*) is to find constraints that structure a model's knowledge representation in theoretically efficacious ways. Given that these constraints succeed for information in a *model* of the mind, their success may be due to the fact that they reflect constraints on information *in the mind itself.* This investigation's search for the optimal representation of procedural knowledge for the model is motivated, in part, by faith in Fodor's research programme.

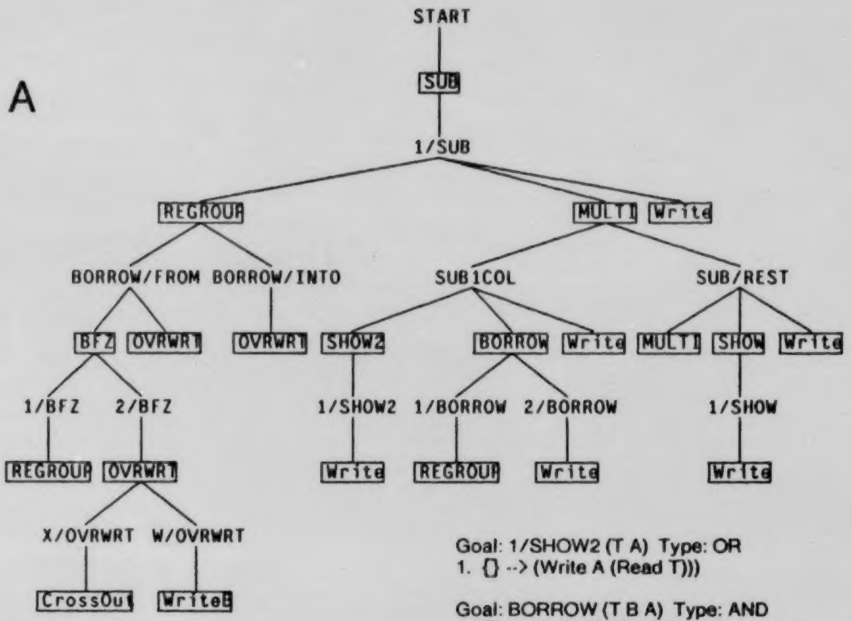
The catch is showing that the success of the model actually depends on the constraints. A proposed constraint on mentalese is not convincingly supported if violating it still allows a successful model to be constructed. The hard part, therefore, is showing that the form of the knowledge representation makes a difference in the model's predictions. This typically requires rather complicated competitive arguments. I was surprised to find as many as I did. Indeed, most of the argumentation in following chapters concerns the representation.

That's enough commentary. Let's move on to the knowledge representation itself. A student's knowledge state is represented by a four-tuple:

1.	*a procedure*:	knowledge about appropriate problem solving actions and their sequence
2.	*a grammar*:	knowledge about the syntax of a mathematical notation
3.	*patches*:	knowledge about past impasses and repairs
4.	*critics*:	knowledge about "wrong" problem states and problem solving actions

The most important of these is the procedure (sometimes called the *core procedure*). Procedures and grammars will be described in this section and two following it. Patches and critics are components of repair theory that won't be described in detail in this chapter.

A procedure is represented as an And-Or graph, or *AOG* (Winston, 1977). Figure 2–6a sketches an AOG for a version of subtraction that will be often used in this document for illustrations. AOG nodes are called *goals*, and links are called *rules*. Rules are directed, and are always drawn running downward. The goals just beneath a goal are called its *subgoals*.


A
START
SUB
1/SUB
REGROUP
MULTI
Write
BORROW/FROM
BORROW/INTO
SUB1COL
SUB/REST
BFZ
OVRWRT
OVRWRT
SHOW2
BORROW
Write
MULTI
SHOW
Write
1/BFZ
2/BFZ
1/SHOW2
1/BORROW
2/BORROW
1/SHOW
REGROUP
OVRWRT
Write
REGROUP
Write
Write
X/OVRWRT
W/OVRWRT
CrossOut
WriteB

B

Goal: START (P) Type: OR
1. {} --> (SUB P)

Goal: SUB (P) Type: AND
1. Let T, B and A be top, bottom and answer of the rightmost column of problem P --> (1/SUB T B A)

Goal: 1/SUB (T B A) Type: OR
1. Regrouping problem format --> (REGROUP T)
2. There is a column to the left of T --> (MULTI T B A)
3. {} --> (Write A (Sub (Read T) (Read B)))

Goal: MULTI (T B A) Type: AND
1. {} --> (SUB1COL T B A)
2. Let NT, NB and NA be the top, bottom and answer of the left-adjacent column to T --> (SUB/REST NT NB NA)

Goal: SUB/REST (T B A) Type: OR
1. There is a column to the left of T --> (MULTI T B A)
2. B is blank --> (SHOW T A)
3. {} --> (Write A (Sub (Read T) (Read B)))

Goal: SHOW (T B A) Type: AND
1. {} --> (1/SHOW T A)

Goal: 1/SHOW (T A) Type: OR
1. {} --> (Write A (Read T)))

Goal: SUB1COL (T B A) Type: OR
1. B is blank --> (SHOW2 T A)
2. T<B --> (BORROW T B A)
3. {} --> (Write A (Sub (Read T) (Read B)))

Goal: SHOW2 (T B A) Type: AND
1. {} --> (1/SHOW2 T A)

Goal: 1/SHOW2 (T A) Type: OR
1. {} --> (Write A (Read T)))

Goal: BORROW (T B A) Type: AND
1. {} --> (1/BORROW T)
2. {} --> (2/BORROW T B A)

Goal: 1/BORROW (T) Type: OR
1. {} --> (REGROUP T)

Goal: 2/BORROW (T B A) Type: OR
1. {} --> (Write A (Sub (Read T) (Read B)))

Goal: REGROUP (T) Type: AND
1. Let NT be the top digit of the left-adjacent column to T --> (BORROW/FROM NT)
2. {} --> (BORROW/INTO T)

Goal: BORROW/INTO (T) Type: OR
1. {} --> (OVRWRT T (Concat (One) (Read T)))

Goal: BORROW/FROM (TD) Type: OR
1. TD is zero --> (BFZ TD)
2. {} --> (OVRWRT TD (Sub1 (Read TD)))

Goal: BFZ (TD) Type: AND
1. {} --> (1/BFZ TD)
2. {} --> (2/BFZ TD)

Goal: 1/BFZ (TD) Type: OR
1. {} --> (REGROUP TD)

Goal: 2/BFZ (TD) Type: OR
1. {} --> (OVRWRT TD (Sub1 (Read TD)))

Goal: OVRWRT (D N) Type: AND
1. {} --> (X/OVRWRT D)
2. Let X be the blank space over D --> (W/OVRWRT X N)

Goal: X/OVRWRT (D) Type: OR
1. {} --> (CrossOut D)

Goal: W/OVRWRT (X D) Type: OR
1. {} --> (WriteB X D)

Figure 2–6

AOG for a correct subtraction procedure acquired from the H lesson sequence.

There are two types of goals: AND and OR. To execute an AND goal, all the subgoals are executed. To execute an OR goal, just one of the subgoals is executed. AND goals are drawn with boxes around their labels. Drawings of AOGs abbreviate goals whenever they appear more than once. For instance, OVRWRT is called from several places in the AOG of figure 2-6a, but its subgoals are drawn only for one of these occurrences. Although abbreviation makes this AOG look like a tree, it is really a cyclic directed graph due to the recursive calls of MULTI and REGROUP.

AOG goals are called *non-primitive* if they have subgoals, and *primitive* if they don't. To avoid clutter, AOG drawings display only non-primitives goals and their subgoals. Only four kinds of primitive goals are allowed:

1. *Primitive actions* cause a change in the current problem state. The only primitive actions used in mathematics are ones that write a given alphanumeric symbol at a given position (`Write` and `WriteB`), or ones that write special kinds of symbols (`CrossOut` puts a slash over a symbol). These three primitive actions are the only ones used in the AOGs for the Southbay experiments.
2. *Facts functions* return a number without changing the problem state. The facts functions used in the Southbay AOGs are `Add`, `Sub`, `Add1`, `Sub1`, `Mult`, `Quotient`, `Remainder`, `One` (which always returns 1), `Zero` (which returns 0), and `Concat` (which concatenates two numbers, e.g., `(Concat 1 4)` returns 14).
3. *Facts predicates* return true or false without changing the problem state. The facts predicates used in the Southbay AOGs are: `LessThan?`, `Equal?`, and `Divisible?`.
4. The primitive function, `Read`, returns the symbol written at a given place. Thus, `(LessThan? (Read T) (Read B))` is true if the digit at the place denoted by T is less than the digit at the place denoted by B.

Primitive goals are, by definition, indecomposable — they have no subgoals. Since Sierra's learner learns by composing goals from subgoals, all primitive goals are necessarily a part of the initial knowledge state, KS_0. However, the initial knowledge state may contain non-primitives as well as primitives. For instance, the initial procedure from which the procedure of figure 2-6 was learned contains the non-primitive goal OVRWRT, which crosses out a symbol and writes another symbol over it.

AOG drawings, such as figure 2-6a, do not indicate several kinds of information. To see this information in Sierra, one merely touchs a goal with the mouse (a pointing device) and the goal's complete definition is printed out. In this document, more cumbersome methods must be used to display goal definitions. Figure 2-6b shows the definitions for the non-primitive goals in the AOG of figure 2-6a. Goals have *arguments*, which have the semantics that a recursive procedure's arguments have in a computer language. For instance, SUB1COL has three arguments, T, B, and A. A goal's rules (i.e., the rules leading from it to its subgoals) are listed in the definition. SUB1COL has three rules. Each rule has a *pattern* and an *action*. Patterns are complex, so their description will be delayed for a moment. (In figure 2-6b, non-null patterns are replaced by English glosses; a null pattern is always matches.) An action is a form, in the Lisp sense, which calls the rule's subgoal. The action may pass arguments to the subgoal, often by evaluating facts functions. For instance, SUB1COL's third rule has `(Write A (Sub (Read T) (Read B)))` as its action. This form calls the goal Write passing it the value of A as its first argument, and a number, roughly T−B, as its second argument. (Throughout this document, T, B and A will stand for the top (minuend), bottom (subtrahend) and answer places in a column.) What this action does is write the difference of the top and bottom digits of a column in the column's answer.

An OR goal's rules are tested in left-to-right order. The first rule whose pattern matches is executed. The learner adds new rules at the left. Hence, the left-to-right ordering convention corresponds to a common conflict resolution strategy in production systems called "recency in long term memory" (McDermott & Forgy, 1978). Because the patterns of OR rules test whether to execute a rule, they are called *test patterns.* Although AND rule patterns have the same syntax as OR rule patterns, they are not used to control which rules are executed. The order of execution of AND rules is fixed: the rules are executed in left-to-right order. AND rule patterns are used to retrieve information in the current problem state so that the information can be passed to the rule's subgoal. AND rule patterns are called *fetch patterns.*

The procedure of figure 2–6 will play a role in illustrations of later sections. It is one of the procedures acquired by traversing the H lesson sequence. It is worth a moment to explain what it does informally. The root goal, `START`, and its subgoal, `SUB`, simply initializes column traversal to start with the units column. `1/SUB` chooses between three subgoals: `MULTI` is for multiple column problems. `REGROUP` is for "regrouping" exercises that don't involve any subtraction at all. This subgoal is left over from learning regrouping separately from multi-column subtraction (i.e., from lesson L3). Normally, `1/SUB` never calls it. The third goal, `Write`, is for single column subtraction problems. The "main loop" of multi-column traversal is expressed by `MULTI` as a tail recursion. `MULTI` calls itself via its subgoal `SUB/REST`. `SUB1COL` processes a column. It chooses between three methods for doing so. If the bottom of the column is blank, it copies the top of the column into the answer via the subgoal `SHOW2`. If the top digit of the column is less than the bottom, it calls `BORROW`. Otherwise, it writes the difference of the two digits in the answer. `BORROW` has two subgoals: `1/BORROW` calls `REGROUP`, and `2/BORROW` just takes the difference in the column and writes it in the answer. `REGROUP` is a conjunction of borrowing into the column that originates the borrow (`BORROW/INTO`) and borrowing from the adjacent column (`BORROW/FROM`). In this procedure, `BORROW/FROM` occurs before `BORROW/INTO`. It would be equally correct to reverse their order, but that is not the way that Heath teaches them. Borrowing into a digit is just adding ten to it. Borrowing from the next column is also easy when its top digit is non-zero: the digit is decremented. If the digit is zero, it calls `BFZ`. `BFZ` regroups, which causes the zero to be changed to ten, then it decrements the ten to nine.

2.4 The representation of grammars

It is obvious that students who can solve mathematical problems must have some understanding of the syntax of mathematical notation. The student's knowledge of the notation's syntax is called a *grammar.* Grammars are formalized as two-dimensional context-free grammars. Figure 2–7 displays a grammar for subtraction notation. The grammar representation language has not been subjected to the careful development that the procedure representation language has. Consequently its conventions are, for the most part, matters of convenience rather than theoretical hypotheses. Nonetheless, it is worth going through the grammar representation just to show what kinds of knowledge need to be represented and to note the few places where critical hypotheses lie. Grammars have two kinds of rules:

1. *Category redundancy* rules have the form `X --> Y` where the right side has just one category. This means that everything that is in category Y is also in category X. Thus, `DIGIT --> 5` means that all 5's are digits. Several category redundancy rules may be abbreviated as one rule by using commas in the right-hand side, e.g., `SIGN --> +, -` means that both + and – are signs. The last six rules of figure 2–7 are category redundancy rules.

```
SIGNED/GRID    --->  SIGN CGRID               ; HORIZ
CGRID          --->  ACOL (ACOL)+ (ACOL)      ; HORIZ
ACOL           --->  COL (DIGIT)              ; VERT BARRED
COL            --->  CELL (%DIGIT)            ; VERT UNBARRED
XNUM           --->  NUM (/NUM)+ /NUM         ; VERT UNBARRED
NUM            --->  DIGIT (DIGIT)+ DIGIT     ; HORIZ

DIGIT          --->  ID/ELT, 2, 3, 4, 5, 6, 7, 8, 9
ID/ELT         --->  0, 1
SIGN           --->  +, -
NUM            --->  DIGIT
CELL           --->  DIGIT, XNUM
PROBLEM        --->  SIGNED/GRID
```

Figure 2-7
A grammar for multi-column addition or subtraction problems

2. *Part-whole rules* have the form X --> Y Z where the right side has two or more categories. Part-whole rules define aggregate categories in terms of their parts. The rule X --> Y Z means that X can be composed of parts Y and Z. Whenever one has a Y and a Z that are situated in the appropriate geometric relationship, one has an X. Part-whole rules bear an annotation, located after a semi-colon, that specifies whether the rule's categories are arranged in a horizontal, vertical or diagonal line. For instance,

    `SIGNED/GRID --> SIGN CGRID ; HORIZ`

    means that a signed grid is composed of a sign followed horizontally by a cgrid (`CGRID` stands for "columnar grid").

There are several biases about mathematical notation that have been built into the grammar formalism. The most important one is the distinction between a tuple and a list. There are two kinds of part-whole rules, called tuple rules and list rules. *Tuple rules* are like ordinary context-free rules in that a rule's left-hand category has exactly the parts mentioned on the right (i.e., `SIGNED/GRID` has exactly the parts `SIGN` and `CGRID`.) *List rules* are for defining sequences of arbitrary length. They have a special format. They have exactly three categories on the right side: `W --> X Y+ Z` means that `X` is the category of the first element of the sequence, `Z` is the category of the last element, and `Y` is the category of the middle elements. The plus sign is what differentiates list rules from tuple rules. Both tuple and list rules mark optional categories by placing them in parentheses. For instance, the list rule

`NUM --> DIGIT (DIGIT)+ DIGIT ; HORIZ`

means that a number (a `NUM`) is at least two digits, with arbitrarily many digits in between. The tuple rule

`ACOL --> COL (DIGIT) ; VERT`

means that an answer-column (an ACOL) is a column (a COL) with an optional digit under it. There are other, minor grammar-writing notations in addition to the tuple/list distinction and optionality.*

Some of these grammar-writing notations are more than just a convenience. They are potential elements of a micro-theory of mathematical syntax. For instance, list rules are included because the tripartite notion of begin-middle-end of a sequence is hypothesized to be highly salient. If list rules are absent, sequential categories can still be expressed using only tuple rules. For instance, a multi-digit number can be expressed by

`NUM --> DIGIT (NUM).`

However, this expression loses the idea that the boundary elements of the sequence, the first and last ones, may be special. Using tuple rules, there is no simple way to indicate, for instance, that the first digit should be non-zero. List rules bias the grammar to express sequences so that the first and last elements are special.

One of the main functions of the grammar is to parse problem states (i.e., interpret them syntactically). A *parse tree* is the grammar's interpretation of a particular problem state. It dictates what groups of symbols are relevant in the current problem state. Figure 2–8 shows a problem state and the parse tree that results when it is parsed with the grammar of figure 2–7. The 18 nodes of that parse tree are essentially the only objects that "exist" in the problem state. It is worth a moment to walk down this parse tree in order to get a feel for how the grammar "views" subtraction problems. By the way, the grammar given in figure 2–7 is the one used in all Sierra's subtraction runs. The whole problem is considered a SIGNED/GRID, which has two parts. The left part is just a minus sign, in this case, although the grammar permits "+" to fill this role as well. The right part of the SIGNED/GRID is a CGRID. The grammar defines CGRIDs as list of ACOLs. In this case, there are three ACOLs, namely the hundreds, tens and units columns. ACOL is short for "answer column" because these are exactly its parts: an answer place and a column. In each of these ACOLs, the answer is a BLK (i.e., blank, which is a dummy category that fills optional constituents) and the column is a COL. A COL has a top part and a bottom part. The bottom can be blank, as in the hundreds column. Usually it is a digit. The top part of a COL is a CELL. CELLs are usually just digits, as they are in all three columns here. However, they can be the kind of symbol groups that results from scratching out a number and writing another number over it (called XNUMs in the grammar).

Notice that the grammar does not define any aggregate objects corresponding to the rows of the problem. Essentially, the grammar says that grouping the symbols into columns is relevant but grouping them into rows is not. From this perspective, the grammar is a skill-specific ontology. It defines the natural kind terms that are relevant for the skill.

---

* To accommodate two-dimensionality, the usual interpretation of constituents for one dimensional (string) grammars is modified slightly: the rectangular region occupied by a constituent may not overlap another constituent's region, nor may a constituent's region include symbols that are not descendents of the constituent. Certain notational devices violate these conventions. As it turns out, these are general devices in mathematics, so ways of handling them have been built into the grammar formalism (as opposed to handling them in individual grammars). These are implemented using special annotations on part-whole rules: Among the categories on the right side of a rule, /X means that X must be crossed out, and %X means that X must not be crossed out. After the semi-colon, BARRED means that the categories in a rule must be separated by vertical or horizontal bars, and UNBARRED means that the rule's categories must not be separated by bars.

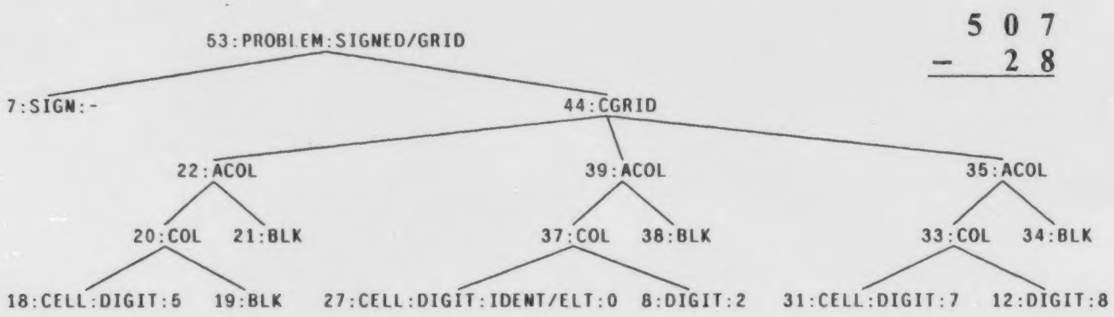


Figure 2–8
A problem state (at upper right) and its parse tree.
Parse nodes are labelled by a unique serial number, followed by the categories of the node, which are separated by colons. For instance, node 18 is a `CELL`, a `DIGIT` and a `5`.

2.5 The representation of patterns

Grammars have an intimate relationship with the patterns that appear on AOG rules. That is why descibing rule patterns has been left until now. A pattern is a set of relations, whose arguments are goal arguments and pattern variables. Patterns do not have logical connectives, quantifiers, equality relations, functions, or other complexities. From a logical standpoint, a pattern is a pure conjunction of literals (a literal is a predicate or a negated predicate), and a pattern variable is interpreted as a Skolem constant. This simplicity is a result of several important constraints on learning that will be discussed in later chapters. In order to illustrate patterns, a version of `SUB1COL` which is slightly different than the `SUB1COL` of figure 2–6 will be used. Its definition, with an English rendition of the rules, is shown in figure 2–9. The first pattern tests whether the bottom (subtrahend) of the column is blank. The second pattern tests whether the top digit of a given column is less than the bottom digit. The third, null pattern is always true. Both goal arguments and pattern variables appear in the patterns. `AC` is `SUB1COL`'s argument. `C`, `T` and `B` are pattern variables. They are of three kinds of relations in patterns:

1. *Categorical relations* are defined by the grammar. For each category in the grammar, a categorical relation is defined. In these patterns, `(COL C)` and `(BLK B)` are the only categorical relations.

2. *Facts predicates* are relations that are defined by the procedure. These were discussed earlier. In these patterns, `(LessThan? T B)` is the only facts predicate.

3. *Spatial relations* are relations that are built into the pattern formalism. There are just six of them:

`(First? S x)`	Object `x` is the first part of some sequential object `S`.
`(Last? S x)`	Object `x` is the last part of some sequential object `S`.
`(Ordered? S x y)`	Object `x` comes before `y` in some sequential object `S`.
`(Adjacent? S x y)`	Object `x` is adjacent to `y` in some sequential object `S`.
`(!Part x y)`	Object `x` is a part of object `y`.
`(Tuple T x y ... z)`	Object `T` is a tuple composed of objects `x`, `y`, `z` etc.

Although the spatial relations are built in, they depend on the grammar for their meaning. For instance, since the grammar defines `COL` to be a vertical category, `(Ordered? C T B)` means that `T` is above `B`. If `COL` were a horizontal category, it would mean that `T` is left of `B`.

```
Goal: SUB1COL (AC) Type: OR

1. {(!Part AC C)              If AC  has a part C,
    (COL C)                   which is of category COL,
    (!Part C B)               with a part B
    (BLK B)                   which is blank,
    ---> (SHOW2 AC)           then call SHOW2 with AC as its argument.

2. {(!Part AC C)              If AC  has a part C,
    (!Part C T)               whose parts are T
    (!Part C B)               and B,
    (Ordered? C T B)          where T is above B ,
    (LessThan? T B)}          and the problem state has a number at location T
    ---> (BORROW AC)          which is less than the number that is at location B,
                              then call BORROW with AC as its argument.

3. {}  ---> (DIFF AC)         Otherwise, call DIFF with AC as its argument.
```

Figure 2–9

Definition for a version of SUB1COL, a goal that processes one column.

Spatial relations, categorical relations and facts predicates are the only relations that patterns may have. There are many reasons for handling relations this way, but chief among them is the so-called *primitives problem.* Any learning theory that describes how knowledge is constructed from smaller units is open to questioning about its set of primitives: what are the units that are assumed to be present when learning begins? If the choice of primitives is left for the theorist to decide, and especially if the theory allows the set of primitives to vary across individuals, then it is usually possible for the theorist to tailor the predictions of the theory to an unacceptable degree by carefully selecting the primitives. Under the approach taken here, the theorist can only vary KS_0, the initial knowledge state, in order to tailor the primitives for individual differences or for different mathematical skills. KS_0 includes the grammar, the primitive facts functions and the primitive facts predicates. The latter are somewhat uncontroversial. The only part of KS_0 worth tailoring is the grammar. Only by modifying the grammar can the theorist manipulate the vocabulary of pattern relations. The vocabulary of primitive relations cannot be manipulated directly. This goes a long way toward dealing with the primitives issue, as chapter 13 shows.

During the execution of a procedure, patterns are matched against the parse tree of the current problem state. However, they are matched differently depending on whether the pattern is a test pattern or a fetch pattern. The patterns that were just used for illustration came from an OR goal, SUB1COL. Therefore, they are test patterns. An OR rule is executed only if its test pattern is true, where truth of a test pattern is defined to be *exact matching*: a pattern matches exactly if all of its relations match. If no rule's test pattern is true, a *halt impasse* occurs (impasses and repairs are described in a later section). The patterns on AND rules (i.e., fetch patterns) have the same syntax as test patterns, but they are used differently. When an AND rule is executed, the fetch pattern is matched to the parse tree, then the bindings of some of its pattern variables are passed to the rule's subgoal. The truth of fetch patterns isn't particularly useful since fetch patterns don't control the course of execution. Fetch patterns are matched using *closest matching:* the matcher uses bindings for the fetch pattern's variables that maximizes the set of relations that match. If more than one such binding exists, an *ambiguity impasse* occurs. Other than the difference in how they are matched, fetch patterns and test patterns have identical syntax and semantics.

2.6 An introduction to induction

Sierra's learner has two components. One is basically an inductive generalization algorithm or *inducer*. The inducer builds a new subprocedure, given a lesson's examples. The other component, called the *deletion unit*, removes one or more rules from the subprocedure that the inducer constructs. The inducer is by far the more complex and important of the two components. Although inducers are common in AI, they are less common than interpreters and problem solvers (which are the main components of Sierra's solver). This section reviews some basic concepts of induction. In the following section, Sierra's learner will be described.

Induction has been defined as the discovery of generalities by reasoning from particulars, or more succinctly, as generalization of examples. As a species of reasoning, induction has been studied in many field under many names. Concept formation, learning by example and grammar inference are just a few of its names. Dietterich and Michalski (1981) and Cohen and Fiegenbaum (1983) review the literature on symbolic (AI) induction. Bierman and Feldman (1972) and Fu and Booth (1975) review the literature on pattern induction and grammatical inference. Anderson, Kline and Beasley (1979) review the literature on prototype formation from an AI perspective. This section introduces some of the basics of induction.

Winston's early work in inductive learning is a classic illustration of induction (Winston, 1975). It will be used throughout this document to furnish simple illustrations of inductive principles. His program learns definitions (concepts) for terms that designate structures made of toy blocks. It does so by examining scenes that have examples of the structure being learned. Figure 2-10 shows some scenes used to teach the concept "arch." When Winston's program compares scene *a* with scene *b*, it discovers that the block that is on top (the lintel) can be either a brick or a wedge. It happens to have a concept, prism, which includes bricks and wedges. It induces that the lintel is a prism. If it later saw an example with a pyramid as the lintel, it would generalize still further, since a pyramid is not a prism. The learner is biased. It is biased toward the most specific generalization that covers the examples. It won't generalize unless it has to. Until it sees a pyramid as the lintel, it will stick with prismatic lintels.

An important distinction is the difference between positive and negative examples. A positive example *is* an instance of the generalization being taught, and a negative example *is not* an instance of the generalization being taught. In the arch-learning illustration, scenes *a* and *b* are positive examples, and scenes *c* and *d* are negative examples. The teacher tells the learner which examples are positive and which are negative. Winston's program made crucial use of *near misses*, negative examples that are almost instances of the target concept. Scene *c* is a near miss. The only thing that prevents *c* from being an arch is the fact that its legs are touching. Scene *d* is not a near miss.

Scene *e* is a near miss that raises an important issue. It was just mentioned that Winston's program had a conservative bias with respect to positive examples. It only generalized if it had to. With respect to negative examples, the program has the opposite bias. In near miss *e*, two relations are missing. The left leg is not supporting the lintel, and the right leg is also not supporting the lintel. Winston's inducer decides that both these support relations are necessary parts of the generalization. A more conservative learner would decide that either left-leg support or right-leg support was necessary, but it wouldn't require that both be present for a structure to be an arch. A conservative learner would accept scenes *f* and *g* as arches, but Winston's program would not. For the conservative learner to learn that both left-leg support *and* right-leg support were necessary, it would have to be given both *f* and *g* as negative examples. So, Winston's learner has two biases: conservative for positive examples, and liberal for near misses. It is important to understand what the biases of an inducer are since they can be critical in making it learn like a human.

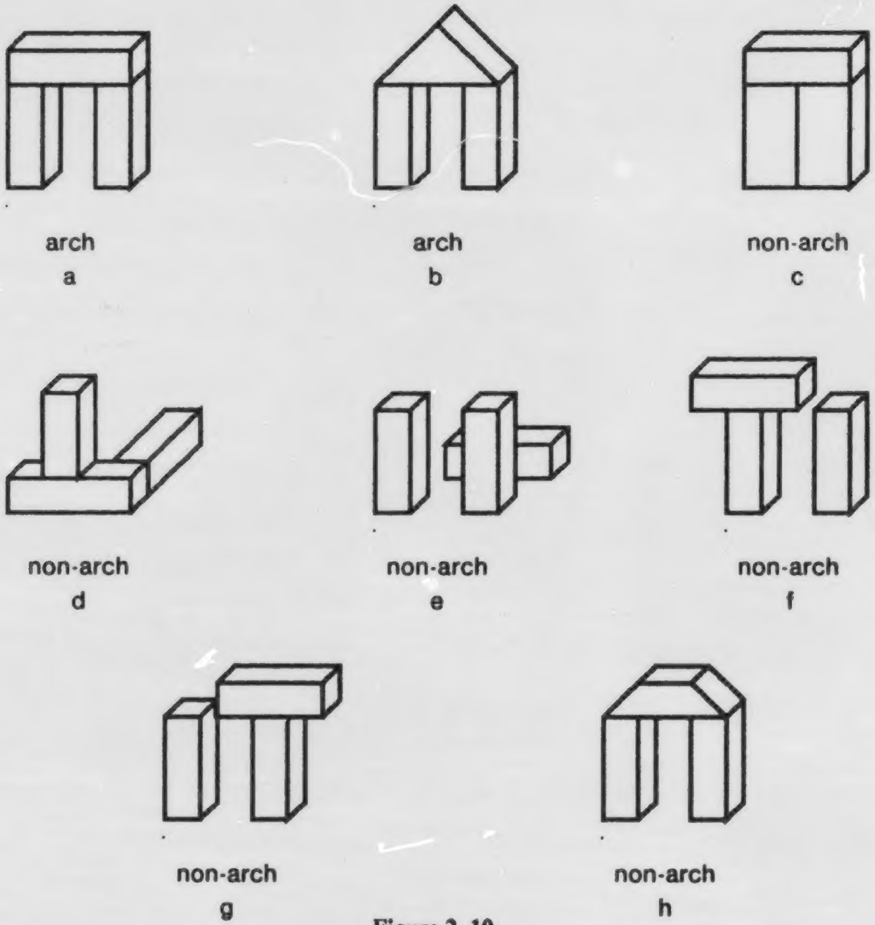


Figure 2–10
Some examples, negative examples and near misses of Winstonian arches.

The concepts of positive and negative examples, near misses and bias have been introduced. Another key concept is the *class of all possible generalizations* that the learner can induce. In most cases, the class of all possible generalizations is determined by a representation language: any expression that can be constructed in the representation language is a possible generalization. Usually, the class is infinitely large. The arch learner's representation language is a certain kind of semantic net language. It uses about a hundred primitive predicates, such as `(ISA x 'WEDGE)` and `(SUPPORTS x y)`. A critical feature of the language is that it does not have disjunction. It has no way to say, for instance, that the lintel is a brick *or* a wedge. Of course, the language could easily have disjunctions added, allowing it to say

```
(OR (ISA x 'BRICK) (ISA x 'WEDGE))
```

or perhaps

```
(ISA x (ANY-OF 'BRICK 'WEDGE)).
```

However, Winston chose not to allow disjunction in the language in order to constrain the class of all possible generalizations, thereby controlling the inducer in an indirect way. This indirect influence plays a tacit role in the treatment of scenes *a* and *b* of figure 2–10. Although the arch learner is biased to take the most specific generalization that covers the examples, it is not allowed to induce that the lintel is a brick *or* a wedge. The representation language forces it to generalize slightly beyond the evidence and decide that the lintel is a prism. Hence, it recognizes scene *h* as an arch, even though it has never seen it before, because the lintel is a prism, a trapezoidal prism. If the language permitted disjunctions, the learner would not make the inductive leap from "brick or wedge" to "prism" and hence would not recognize scene *h* as an arch. In short, the constraints on the class of all possible generalizations, which are usually determined by the representation language, exert a crucial control over the character of the inducer's learning. To reiterate: the two major determinants of induction are the biases of the inducer and the constraints on its possible generalizations.

The ideas that have been introduced can be summarized as follows: The input to an inducer is a sequence of examples; the output is a set of expressions in some representation language such that (1) each expression is *consistent* with all the examples, and (2) the set of expressions is maximal respect to a certain partial order, called a *bias* (or a simplicity metric, or weighting function). That is, the output consists of the simplest expressions in the representation language that are consistent with all the examples.

The definition of consistency and bias varies with the task. For instance, suppose the task is to induce grammars from strings. The examples are strings, and the task is to induce expressions (grammars) in some specified grammar-representation language. A grammar is consistent with a string if the grammar parses it (or more formally, the string is in the language generated by the grammar). A typical bias is to prefer simple grammars (e.g., fewest rules, or fewest non-terminal categories). A bias based on counting rules or categories would be a total order, since any two grammars can be compared. In general, biases are partial orders rather than total orders. For instance, suppose the bias is to prefer grammar A over grammar B whenever A's rules are a strict subset of B's rules. This means that certain pairs of grammars will be incomparable: neither may be a subset of the other. When the bias is a partial order, more than one grammar may be maximal. That is why the output of an inducer is defined to be a *set* of expressions, not just a single expression. A last comment is that bias is applied *after* consistency, so to speak. In effect, the inducer first finds all abstractions consistent with the examples, then it finds the maximal elements of this set.

Negative examples and discrimination examples

Negative examples are very important. The previous discussion of near misses indicated how important they were for Winston's inductive learner. Another important use of negative examples is to recover from overgeneralizations. Suppose the target concept is more specific than the concept that the inducer has at the moment (e.g., the target is `PRISM` but the inducer has guessed `BLOCK`). A negative example can be used to force the inducer to make its guess more specific (e.g., showing the inducer a negative example that is a block and not a prism). No positive example could force the inducer to retreat from the overgeneralization in this way (Gold, 1967). A critical issue for this theory is whether instruction in mathematical skills uses negative examples, and if so, how.

Strictly speaking, a negative example of a mathematical procedure is a problem state sequence that illustrates an *incorrect* way to solve the problem. In a textbook, such a worked exercise might be labelled, "This is a wrong way to do subtraction problems. Do not use this way." I have never seen such examples in textbooks. However, negative examples do occur in classrooms under several circumstances. When a teacher has the class solve a problem on the blackboard, incorrect problem state sequences will sometimes be generated (or partially generated before the teacher stops the student solving the problem). These incorrect solutions are negative examples. They may even qualify as near misses. A similar situation occurs when students are doing seatwork. Students having difficulty often ask for the teacher's help. The teacher may watch them solve a problem, then point out where the student went wrong. Such incorrect solutions also serve as negative examples. So, negative examples are not absent in normal instruction. But they are not common, and they are not used in any methodical way.

There is another kind of example which functions as a negative example in certain ways, although it is not, properly speaking, a negative example. Solving a problem that doesn't require a certain subprocedure provides a negative example for induction of the subprocedure. I believe the traditional name among curricula writers for such examples is *discrimination* examples. A discrimination example is one that demonstrates when *not* to use the subprocedure that is being taught in the current lesson. For instance, an example that doesn't borrow is a discrimination example when it appears in the midst of a borrow lesson. Such an example can help the inducer discriminate the conditions that determine when one should borrow by providing *negative instances* of borrowing (i.e., problem states when one should not borrow). With regards to the induction of the test pattern that governs a new subprocedure, negative instances act as negative examples. So a discrimination example provides a negative instance of a certain subprocedure's test pattern, but it is not a negative example.

At first glance, it seems that some textbooks provide discrimination examples and some don't. This is rather odd. If induction is indeed what students do, and given that induction can proceed more efficiently when negative instances are available, then it is amazing that some curricula omit discrimination examples. A closer examination of the textbooks in question reveals that they actually do have discrimination examples. However, the discrimination examples for a certain subprocedure do not occur in the introductory lesson on the subprocedure. Instead, they are placed later. Often they appear in review lessons. Another place they appear is in lessons of subsequent subprocedures. For instance, discrimination examples for simple, non-zero borrowing are provided by the examples used to introduce borrowing from zero. In

$$\begin{array}{r} 5\;\,9\;\;\; \\ 6^{1}0^{1}7 \\ -\;2\;\,3\;\,8 \\ \hline 3\;\,6\;\,9 \end{array}$$

subtracting the tens column always provides a negative instance for borrowing. By the time the solver gets to processing it, the tens column's top digit has been changed to 9, so the column never requires a borrow. This can be used by the inducer in inferring that $T<B$ is the correct condition for borrowing. So an ordinary positive example of borrowing-from-zero necessarily provides a discrimination example for the subprocedure of borrowing. In certain cases, the same example can be both a positive example and a discrimination example for a certain subprocedure. In short, it appears that discrimination examples are present, one way or another, although they may occur late in the lesson sequence.

At the present time, Sierra's learner is not able to use discrimination examples for a subprocedure unless they occur in the lesson where the subprocedure is introduced. Consequently, when a textbook's content is formalized as a lesson sequence, discrimination examples are moved forward in the sequence. For instance, example *b* in figure 2–4b is a discrimination example that is not in the corresponding real lesson. It came from a review lesson that appears a little later in the textbook. Appendix 7 discusses how this technical limitation will be removed in later versions of Sierra.

2.7 The learner

A typical inducer's task is completely specified by (1) the representation language, (2) the definition of *consistency*, and (3) the definition of *bias*. Sierra's learner is atypical in that its specification involves one further constraint: it may add *at most one subprocedure per lesson*. This constraint is a fourth kind of constraint. It is essentially an upper bound on the *rate* at which the inducer may change its candidate generalizations.

As far as I know, Sierra's learner is the first AI inducer to use a rate constraint. Rate constraints might be profitably exploited in other applications, such as the knowledge acquisition phase of knowledge engineering. In fact, a quick survey of the knowledge acquisition literature reveals an amusing "hole." There is a great deal of complaining about the so-called knowledge acquisition bottleneck. It is hard to get human experts to formalize their expertise as e.g., production rules. One often heard solution is to have the system learn the knowledge on its own, e.g., by discovery or by analogy. However, few human experts acquired their knowledge this way. Most of them didn't discover their knowledge or infer it, they learned it in school or from a mentor. The "hole" in the knowledge acquisition research is that no one, to my knowledge, is trying to get their expert system to learn like human experts learn. Such a system would take advantage of the structure that its mentor places on the instruction. The present research, in its explication of felicity conditions, should be helpful in building such a knowledge acquisition system. Presumably, such a system will be easier for human experts to educate than present systems. Because many experts are experienced teachers, they are more familiar with formatting their knowledge as lesson sequences than as production rules. Felicity conditions might help solve the knowledge acquisition bottleneck. Alas, this research is not aimed at such practical (and potentially lucrative) goals. Its aims are merely scientific.

Representation language, the first of the constraints on Sierra's learner, has been defined already. This section is devoted to defining the others. They are discussed in the following order: consistency, subprocedure and bias. The actual algorithm used to implement the inducer is not discussed here because any algorithm that meets the specifications would do just as well from the standpoint of the theory.

The definition of consistency

A procedure is *consistent* with an example if and only if its solution to the example's problem is exactly the same problem state sequence as the example itself. This definition captures a controversial felicity condition. Teachers guarantee that any procedure that always produces a correct problem state sequence will be acceptable. It matters less what students say or think; they are evaluated on what they *do*. Consequently, in order to succeed in school, students need only induce a procedure that is *consistent with respect to the problem state sequences* of the teachers' examples. It's rational that students would take the simplest, most efficient road to success. In fact, they do. Induction from problem state sequences is just what students seem to do. The felicity condition captures this whole complex: the teachers' guarantee, the way the guarantee simplifies learning, and the fact that students actually take advantage of the guarantee by using the simplified way to learn. This felicity condition is labelled the induction hypothesis in chapter 3, where it is formally defined.

The definition of subprocedures

The maximum amount of material that may be added to an AOG by the learner during a lesson is called a *subprocedure*. In Lisp terms, a subprocedure is like one clause from a COND statement: it's a new conditional branch that consists of a sequence of several steps, where each step calls existing code. If procedures are presented as augmented transition nets or ATNs (Woods, 1970; Winston, 1977), then a new subprocedure is a new arc and a new level that is called by the new arc (see figure 2-11). In AOG terms, a subprocedure consist of several components:

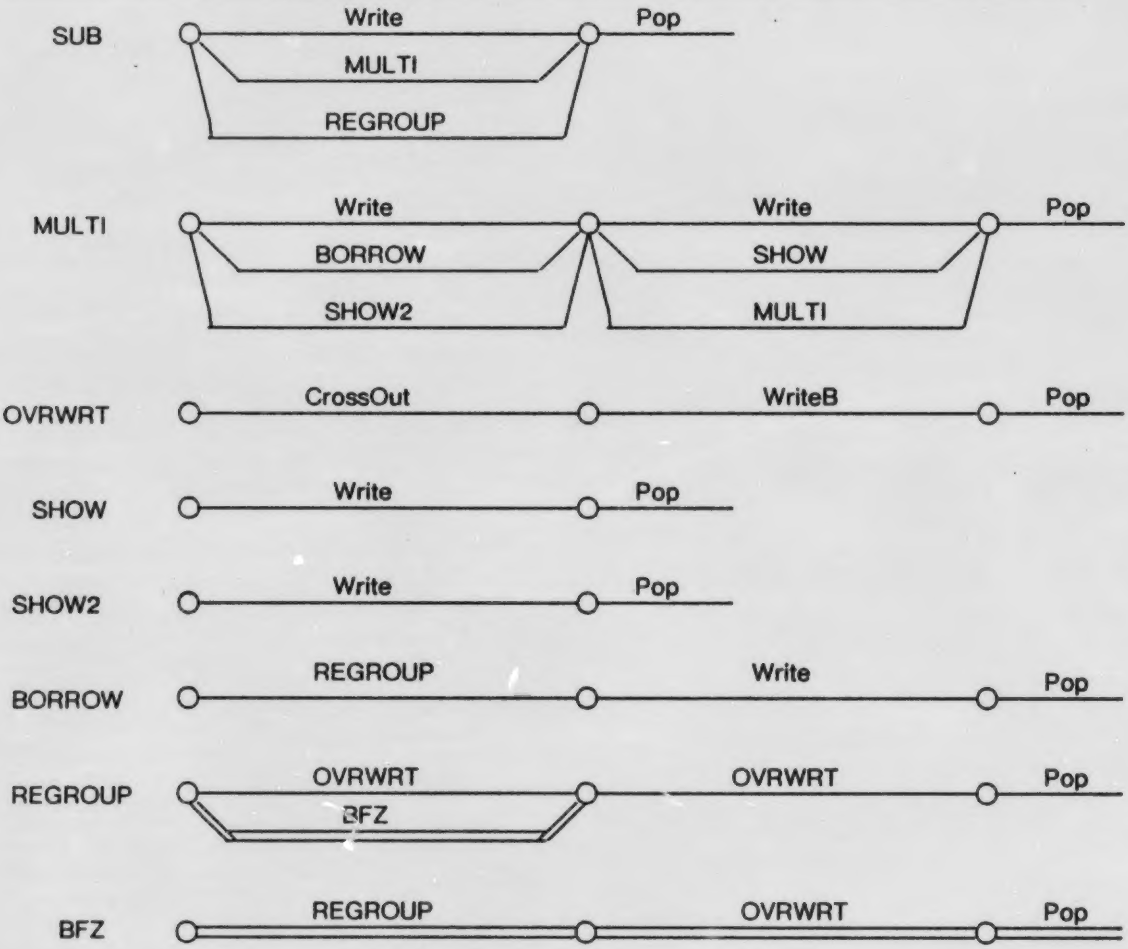

SUB
Write
Pop
MULTI
REGROUP
MULTI
Write
Write
Pop
BORROW
SHOW
SHOW2
MULTI
OVRWRT
CrossOut
WriteB
Pop
SHOW
Write
Pop
SHOW2
Write
Pop
BORROW
REGROUP
Write
Pop
REGROUP
OVRWRT
OVRWRT
Pop
BFZ
BFZ
REGROUP
OVRWRT
Pop

Figure 2-11
The procedure of figure 2-12b drawn as an ATN, where arcs run left to right.
The new subprocedure's arcs are shown with double lines.

1. a new rule, called the *adjoining rule*, that is added to an existing OR goal, called the *parent OR*.
2. a new AND goal, called the *new AND*. The adjoining rule calls it.
3. one or more new OR goals, called the *trivial ORs*. The trivial ORs are called by the new AND's rules. Each trivial OR has a single trivial (i.e., null pattern) rule that calls some existing AND goal. These existing AND goals are called the *kids*.

Figure 2–12 illustrate these components of a subprocedure by showing an AOG before and after a subprocedure has been added. This subprocedure, by the way, is the one acquired from the lesson of figure 2–4, teaches how to borrow from zeros. It will be used throughout this section as a running example. The pre-lesson AOG (figure 2–12a) can borrow only from non-zero digits; the post-lesson AOG (figure 2–12b) can borrow across zeros. `BORROW/FROM` is the subprocedure's parent OR. The adjoining rule connects `BORROW/FROM` to `BFZ`. The new AND is `BFZ`. The trivial ORs are `1/BFZ` and `2/BFZ`. The kids are `REGROUP` and `OVRWRT`.

The reason subprocedures have the particular structure that they do is the subject of lengthy argumentation, which will be presented in following chapters. To summarize that argumentation, a certain felicity condition, one-disjunct-per-lesson, mandates that just one new choice be introduced into the procedure's structure. This choice is created by adding the adjoining rule to the parent OR. This means that there is now a new way to achieve that goal. A choice has been added. For convenience, all places where there could eventually be choices, but so far there are none, are marked structurally. Thus, the subgoals of the new AND are created as trivial ORs. Trivial ORs provide a place for later subprocedures to attach. In fact, this subprocedure's parent OR, `BORROW/FROM`, was created as a trivial OR for `REGROUP`.

The definition of bias

Given a lesson and a procedure, the learner first generates all possible subprocedures that make the procedure consistent with the lesson's examples, then it uses *bias* to define the maximal subprocedures. (Actually, the algorithm is more complex, but the effect is the same.) Bias is defined by several ordering predicates. Each bias predicate will be stated and discussed in turn. A>B will be used to indicate that the bias prefers procedure A over procedure B.

> *Maximally general test patterns*
> If two subprocedures, A and B, are equal in every way except that A's adjoining rule's test pattern is a subset of B's adjoining rule's test pattern, then A>B.

The adjoining rule of a subprocedure is an OR rule, so it's pattern is a test pattern. It controls when the subprocedure will be executed. For instance, in figure 2–12, the adjoining rule connects `BORROW/FROM` to `BFZ`. If its test pattern is true, `BORROW/FROM` calls `BFZ`; if it is false, `BORROW/FROM` calls a subgoal that simply subtracts one from `BORROW/FROM`'s argument. For this test pattern to be consistent with the examples, it should be true in all problem states where `BORROW/FROM` is the current goal and the subprocedure is invoked by the teacher. Such problem states are called *positive instances*. It should be false in problem states where `BORROW/FROM` is the current goal and the teacher did not invoke the subprocedure. Such states are called *negative instances*. Given the lesson of figure 2–4, the positive instances are states *a* and *b* below, and negative instances are states *c* and *d*.

```
                                            1 14
a.   3 0 4     b. 7 0 7     c. 8 2 4     d. 8 2¹4
   - 1 2 6      -   2 8      - 3 5 8      - 3 5 8
   -------     --------     --------     --------
                                                6
```

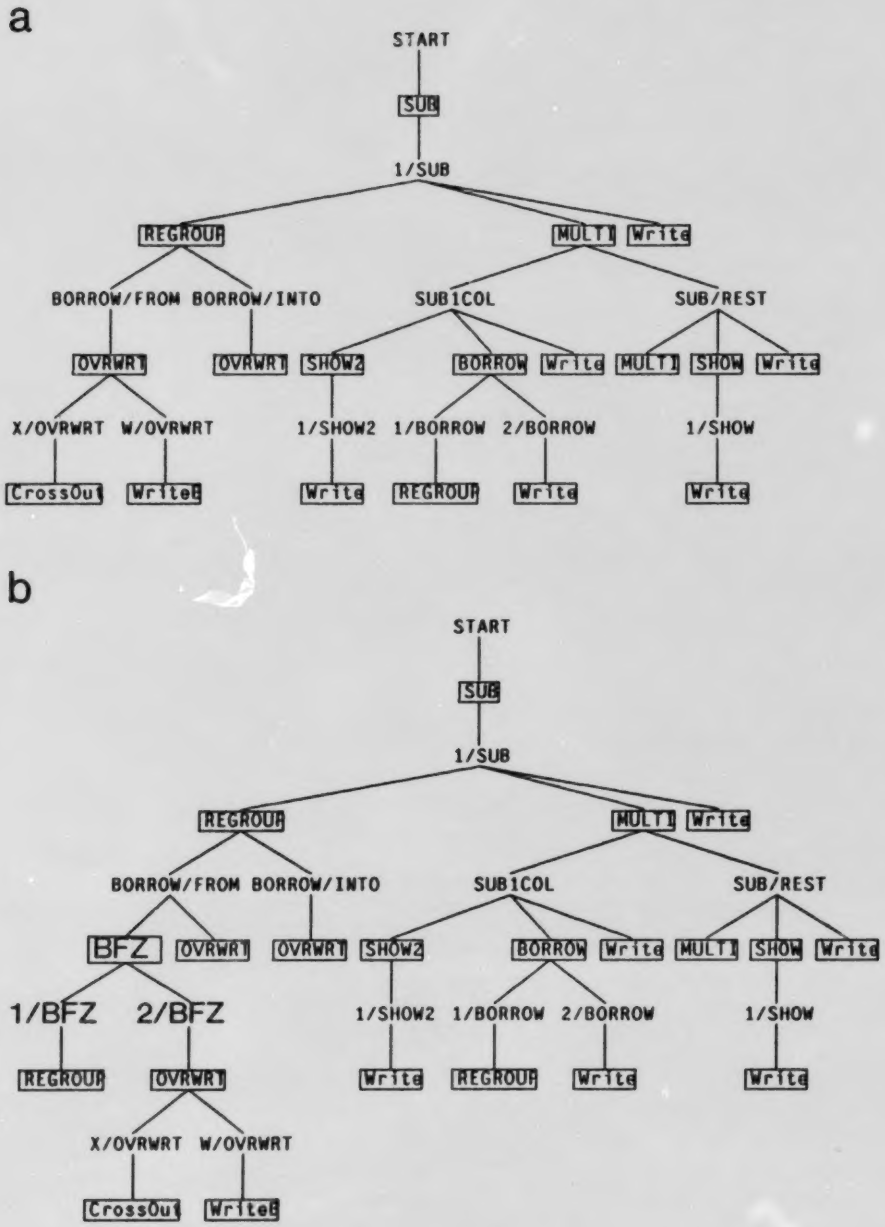

a
START
SUB
1/SUB
REGROUP
MULTI
Write
BORROW/FROM
BORROW/INTO
SUB1COL
SUB/REST
OVRWRT
OVRWRT
SHOW2
BORROW
Write
MULTI
SHOW
Write
X/OVRWRT
W/OVRWRT
1/SHOW2
1/BORROW
2/BORROW
1/SHOW
CrossOut
WriteB
Write
REGROUP
Write
Write
b
START
SUB
1/SUB
REGROUP
MULTI
Write
BORROW/FROM
BORROW/INTO
SUB1COL
SUB/REST
BFZ
OVRWRT
OVRWRT
SHOW2
BORROW
Write
MULTI
SHOW
Write
1/BFZ
2/BFZ
1/SHOW2
1/BORROW
2/BORROW
1/SHOW
REGROUP
OVRWRT
Write
REGROUP
Write
Write
X/OVRWRT
W/OVRWRT
CrossOut
WriteB

Figure 2–12
AOGs before (a) and after (b) H's lesson on borrowing from zero.
The new subprocedure's goals are shown in a larger font.

There are often millions of consistent test patterns. For illustration of how the bias affects the induction of this test pattern, however, we'll consider just these four patterns, not all of which are consistent (each pattern is followed by its English translation):

1.	{}	Always true.
2.	{(0 TD)}	BORROW/FROM's argument is a zero.
3.	{(Part! C TD)) (Part! AC C)) (Part! G AC)) (Part! G X)) (Adjacent? G AC X)) (Last? G X)})	BORROW/FROM's argument TD is in a column, AC, that is adjacent to the rightmost column in the problem, X. That is, BORROW/FROM's argument TD is in the tens column.
4.	{(0 TD) (Part! C TD)) (Part! AC C)) (Part! G AC)) (Part! G X)) (Adjacent? G AC X)) (Last? G X)})	BORROW/FROM's argument TD is a zero, and it is in a column, AC, that is adjacent to the rightmost column in the problem, X. That is, TD is a zero and it is in the tens column.

Pattern 1, the trivial pattern, is true of both positive instances (indeed, it is always true). But it is not false of the negative instances. Hence it is not consistent with the examples. Pattern 2 is true of both positive instances and false of the negative ones. It is consistent. Moreover, since the only pattern that could be a proper subset of it is {}, which is inconsistent, pattern 2 is maximally general. It is accepted by the bias. Pattern 3 is true of the positive instances but it is not false of one of the negative instances (*c*). Hence pattern 3 is inconsistent. When it is conjoined with pattern 2, the result, pattern 4, is consistent. But it is not maximally general because it contains pattern 2 as a proper subset, and pattern 2 is consistent. Pattern 4 is not accepted by the bias. The bias prefers pattern 2 instead. To put it intuitively, pattern 4 would represent students who believe that they should only borrow-from-zero for zeros that are in the tens column. Such a belief would appear in the students' work as a bug. But no such bug has been observed. In order to account for this fact and many others, the theory adopts a bias toward maximally general test patterns. We move on to the next bias predicate.

> *Lowest parent*
> Given two subprocedures, A and B, for possible addition to a procedure P, if A is *lower* than B in that there is a path from P's root to A that passes through B, then A>B.

The best way to understand the lowest parent bias is to see an example. Figures 2–12b, 2–13c and 2–13d show three AOGs corresponding to adding different subprocedures to an initial AOG. The initial AOG is shown in figure 2–12a. All three procedures are consistent with the lesson's examples. However, subprocedure 2–13c's parent, SUB1COL, is higher than subprocedure 2–12b's parent, BORROW/FROM. Subprocedure 2–13d's parent, 1/SUB, is higher still. The lowest parent bias prefers 2–12b. Essentially, 2–12b represents the idea that the new subprocedure, BFZ, is a kind of borrowing. 2–13c represents the idea that BFZ is a way to process columns (i.e., there are three kinds of columns: easy, non-borrow columns; harder, borrowing columns; and super hard, borrow-from-zero columns). 2–13d represents the idea that BFZ is a way to process a whole problem. (i.e., there are two kinds of problems: regular problems and problems that require BFZ.)

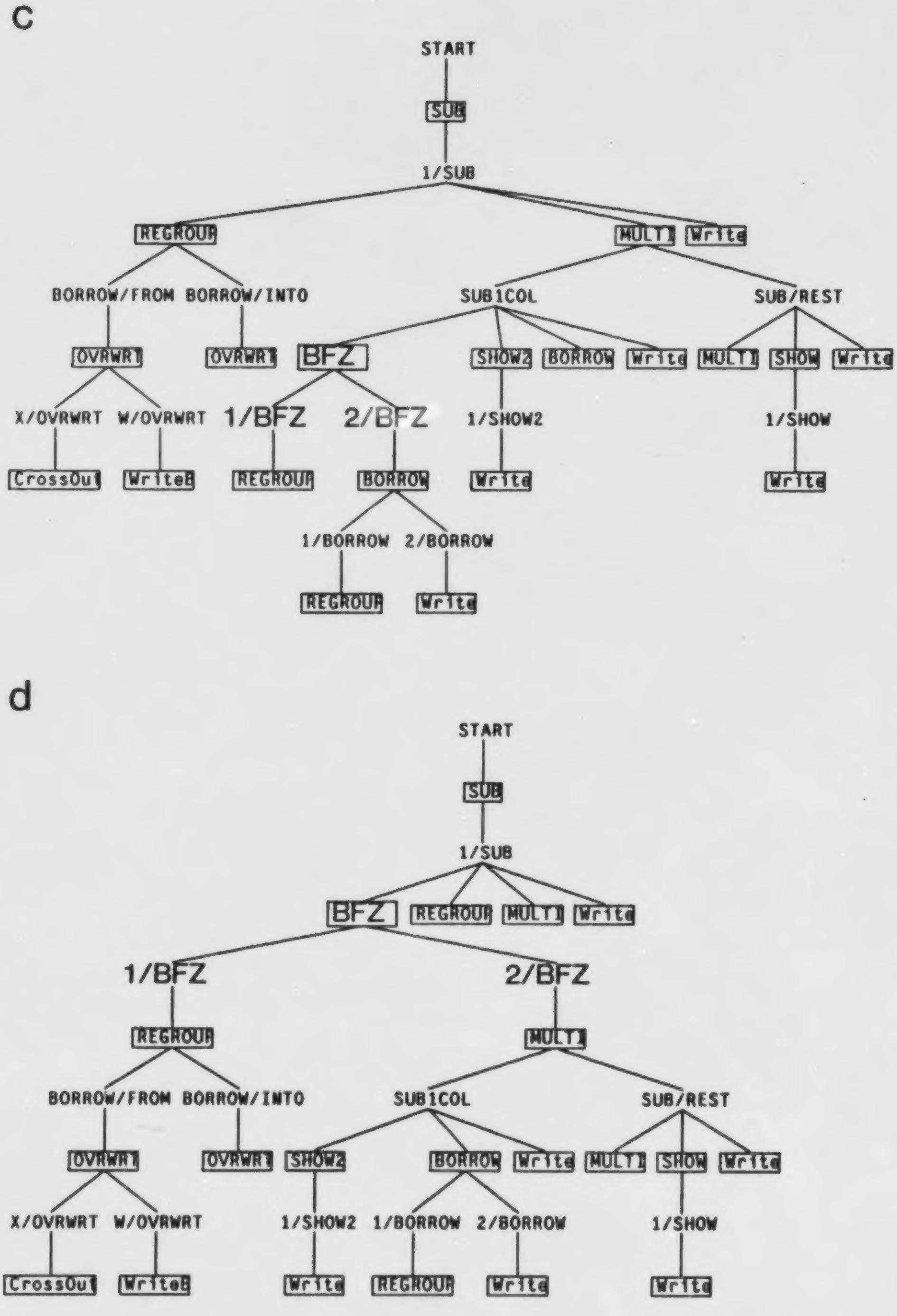

c
START
SUB
1/SUB
REGROUP
MULTI
Write
BORROW/FROM
BORROW/INTO
SUB1COL
SUB/REST
OVRWRT
OVRWRT
BFZ
SHOW2
BORROW
Write
MULTI
SHOW
Write
X/OVRWRT
W/OVRWRT
1/BFZ
2/BFZ
1/SHOW2
1/SHOW
CrossOut
WriteB
REGROUP
BORROW
Write
Write
1/BORROW
2/BORROW
REGROUP
Write
d
START
SUB
1/SUB
BFZ
REGROUP
MULTI
Write
1/BFZ
2/BFZ
REGROUP
MULTI
BORROW/FROM
BORROW/INTO
SUB1COL
SUB/REST
OVRWRT
OVRWRT
SHOW2
BORROW
Write
MULTI
SHOW
Write
X/OVRWRT
W/OVRWRT
1/SHOW2
1/BORROW
2/BORROW
1/SHOW
CrossOut
WriteB
Write
REGROUP
Write
Write

Figure 2–13
Subprocedures, in large font, that are attached to higher parents.

There is a somewhat different perspective on the lowest parent bias: the lower the parent, the more general the subprocedure. For instance, 2–12b and 2–13c can cope with four column borrow-from-zero problems, such as *a* below,

$$\text{a.}\quad \begin{array}{r} 5\;0\;7\;2 \\ \underline{-\;1\;1\;9\;1} \end{array} \qquad\qquad \text{b.}\quad \begin{array}{r} 5\;0\;0\;2 \\ \underline{-\;1\;1\;1\;9} \end{array}$$

but 2–13d can not because its subprocedure is attached *above* `MULTI`, the loop across columns. 2–12b can borrow from multiple zeros, as in *b*, but neither 2–13c nor 2–13d can. So the lowest parents bias is a bias toward increasing the applicability of the new subprocedure.

This bias has a certain elegant relationship to the test pattern bias. Note, first of all, that test patterns and parent ORs are two aspects of the same thing: The test pattern expresses *external* conditions on when to call the new subprocedure, and the parent OR expresses *internal* conditions on when to call it, i.e., what goal must be current in order to call it. (In ordinary production systems, test patterns and parent ORs would be syntactically indistinguishable because they would both be conditions in the left-hand side of a rule.) Given this duality, their respective biases ought to be the same, and they are. The test pattern bias is toward maximizing the applicability of subprocedure. The lowest parent bias is also towards maximizing applicability. To put it a little differently, these two biases both say that students would prefer to risk errors of commission (i.e., executing the new subprocedure when it really shouldn't be executed) rather than risk errors of omission. On to the next bias predicate.

> *Maximally specific fetch patterns*
> If two subprocedures, A and B, are identical except for their fetch patterns, and each fetch pattern on A's new AND's rules is a superset or equal to the corresponding fetch pattern in B, then A>B.

This bias prefers the largest, most specific patterns for fetch patterns. This makes sense, given the role that fetch patterns play. The basic problem that a fetch pattern solves is deciding which of the many visible objects (e.g., which digit or which column) a subgoal should use as its arguments. In order to maximize the fetch pattern's power to discriminate, the learner remembers everything about the lesson's examples that might prove useful in fetching — it remembers maximally specific patterns. It does so in order that problem solving can approximate the lesson situation as closely as possible. If some idiosyncracy of the lesson's examples is stored, no harm is done. Although the idiosyncratic relations won't match during problem solving, fetch patterns are matched closely rather than exactly, so the fetch will succeed anyway.

Besides inducing patterns, the learner builds actions for each of the new rules. This sometimes involves inducing nests of functions. Functions are typically needed whenever the worked example introduces a new number, a number that is not equal to one of the numbers visible in the current problem state. These new numbers are usually the result of some facts function that is performed invisibly by the teacher (or textbook). In the first example of the lesson (see figure 2–4a), a 9 is introduced in problem state *d*. Some possible candidates for the function nest that generates the 9 follow (English presentations have been substituted for the pattern variables that would normally appear as the arguments of Read):

```
1. (QUOTE 9)
2. (Add (Read <original top digit of hundreds>) (Read <bottom of units>))
3. (Sub1 (Add (Read <top of tens>) (Read <original top of tens>)))
4. (Sub1 (Read <top of tens column>))
```

Function nest 1 is a constant. It turns out to be a correct function nest. Function nest 2 will be filtered out by the lesson's next positive example, 707−28, because 7+8≠9. The third nest will never be filtered out by any example since the subprocedure is only called when the top of the tens column is zero. However, this function nest is ruled out by the show-work principle. The show-work principle is a felicity condition that states that examples of a new subprocedure are expected by the student to show all their work. What this means, in practice, is that facts functions won't be nested by Sierra's learner. To do so, as in the third nest above, is to hide an intermediate result instead of writing it down. Apparently, students don't believe that the teacher will do that, so they never bother to consider nested facts functions.* The fourth function nest is logically equivalent to the first. It too is consistent with all the lesson's examples. However, the use of `Sub1` instead of the constant 9 has a subtle effect on local problem solving, which allows one to detect which one students prefer. They prefer the constant. The following bias expresses that preference and others like it:

> *Smallest arity*
> If two subprocedures, A and B, are identical except for a function nest, and the arity of A's nest is smaller than the arity of B's nest, then A>B, where the arity of a function nest is the sum of the number of argument places in its functions (i.e., constants and nullary functions count 0, unary functions count 1, binary functions count 2, etc.).

This is a rather minor bias that has a clear intuitive interpretation. Suppose that executing binary facts functions requires greater use of cognitive resources than executing a unary facts function. The bias then means that students prefer function nests which reduce their cognitive load during execution. There are just two more bias predicates left to discuss.

> *Fewest kids*
> If two subprocedures, A and B, have the same parent OR, and A has fewer kids than B, then A>B.
>
> *Lowest kids*
> If two subprocedures, A and B, have the same parent OR and the same number of kids, and each of A's kids is lower than or equal to the corresponding kid in B, then A>B, where "lower" is defined as in the lowest parent constraint.

These last two biases were discovered by trial and error. Although they are needed in order to improve the theory's predictions, I have, as yet, only a speculative interpretation for them, which is discussed in chapter 19. What makes these biases confusing is that they are opposing biases. Chapter 19 shows how the fewest-kids bias increases the generality of the subprocedure while the lowest-kids bias decreases it.

---

* We can infer this by relaxing the show-work principle and seeing if the resulting predictions are accurate. If the show-work principle is relaxed slightly so that facts functions can be nested one deep, then approximately 450,000 distinct function nests are induced for this lesson. Many of them lead to star bugs.

A summary of the learner's operation

All the criteria for the learner have been defined. The representation language and consistency were defined. Rate constraints were defined by defining subprocedures. The biases were defined. Any algorithm that satisfy these specifications would serve as an implementation for Sierra's learner*. Sierra's actual algorithm is not far from a brute force algorithm. However, it uses some tricks to reduce the computational resources required. For instance, it uses a version space to represent the set of all consistent test patterns (see section 18.1 and Mitchell, 1982).

It was mentioned earlier that Sierra's learner has a second, minor component, called the deletion unit, that deletes one or more rules from the subprocedure(s) produced by the inducer. The deletion unit's operation is quite simple. Suppose the inducer has just produced a subprocedure whose new AND has *n* rules. The deletion unit produces 2^n-2 new subprocedures, one for each non-trivial subset of the AND rules. If the new AND has two rules, R1 and R2, then the deletion unit produces two new subprocedures. One has a new AND with just R1. The other has a new AND with just R2. Chapter 7 discusses why deletion must be a part of the learner. The bottom line is that several observed bugs can't be generated without it.

2.8 Core procedure trees for the Southbay experiment

In order to illustrate the way the learner's inducer works and to start the discussion of observational adequacy, this section illustrates the inducer's performance when given a particular subtraction lesson sequence, the one called H in section 2.2. Sierra's inducer is one-to-many in that it may produce more than one output procedure from a single input procedure and a lesson. This comes out clearly in figure 2–14, which shows the *core procedure tree* for the learner. The core procedure tree shows which procedures are derived from which other procedures. The initial procedure is at the top. It is called "1c" because it can only do one-column problems. The links in the core procedure tree are labelled with the lesson names. Thus, lesson L_1 produces procedure 2c-full. The remainder of this section is a "walk" down the core procedure tree.

The first lesson, L_1, teaches how to solve problems of the form NN−NN. The resulting procedure, 2c-full, can do two column problems, where both columns are "full." The new AND of the subprocedure introduced by this lesson is labelled `MULTI` in figure 2–6, which is the AOG from the procedure labelled "ok" in the core procedure tree. Henceforth, the new AND's names will be indicated in square brackets so that the reader can follow along in figure 2–6. Lesson L_2 teaches how to solve incomplete tens columns, producing a procedure called 2c that can do any two column problem that does not require borrowing [`SHOW`]. Lesson L_3 introduces regrouping offline, so to speak [`REGROUP`]. It uses examples that are not subtraction problems. The subtraction procedure that results from this lesson, 2c-regroup, can do both regrouping exercises and two-column subtraction problems, but it cannot do two-column subtraction problems that require borrowing. That capability is taught by lesson L_4.

---

* There are interactions among the biases, so they must be applied in the following order: lowest parent, fewest kids, lowest kids, smallest arity, and maximally specific fetch patterns. The bias for maximally general test patterns must be applied after the lowest parent bias, but it is independent of the others.

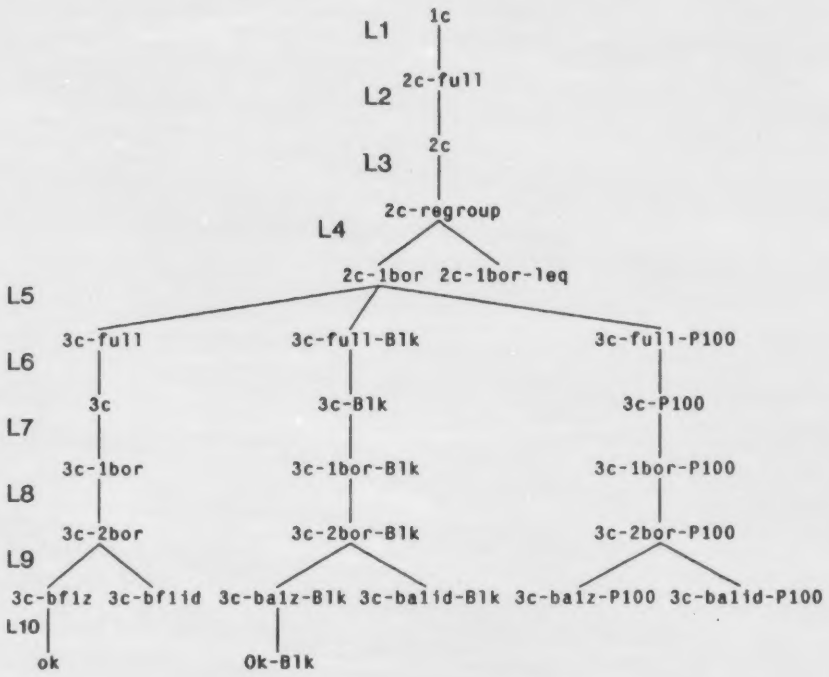

L1
1c
L2
2c-full
L3
2c
L4
2c-regroup
L5
2c-1bor
2c-1bor-leq
L6
3c-full
3c-full-Blk
3c-full-P100
L7
3c
3c-Blk
3c-P100
L8
3c-1bor
3c-1bor-Blk
3c-1bor-P100
L9
3c-2bor
3c-2bor-Blk
3c-2bor-P100
3c-bflz
3c-bflid
3c-balz-Blk
3c-balid-Blk
3c-balz-P100
3c-balid-P100
L10
ok
Ok-Blk

Figure 2-14
The core procedure tree of lesson sequence H, with lesson labels on the left.

Lesson L_4 integrates regrouping into the column-traversal algorithm [`BORROW`]. Notice that there are two output procedures, 2c-1bor and 2c-1bor-leq. Part of what lesson L_4 teaches is *when* to borrow. It uses examples like $34-18$ to show when to borrow (positive examples), and examples like $34-13$ to show when not to borrow (discrimination examples). However, lesson L_4 does not include examples like $34-14$, where the units column's digits are equal. Hence, the learner has no way to tell whether the test for borrowing should be $T<B$ or $T\leq B$. Sierra's learner thus produces two procedures: procedure 2c-1bor borrows when $T<B$, and procedure 2c-1bor-leq borrows when $T\leq B$. This constitutes a prediction that some students will borrow when $T=B$, as in $34-14$. This is a correct prediction. The corresponding bug, which is called N−N-Causes-Borrow, has been observed.

Lesson L_5 teaches how to solve three column problems [recursive call to `MULTI`]. It produces three procedures. Procedures 3c-full and 3c-full-Blk are almost identical. The only difference is the test pattern that they use to tell when to recurse. For procedure 3c-full, the test is whether the current column is the leftmost column in the problem; if it is not, then the procedure recurses. For 3c-full-Blk, the test is whether there are any unanswered columns left. Both procedures lead ultimately to correct subtraction procedures (the ones labelled ok and ok-Blk). The intermediate "Blk" procedures, however, generate star bugs. Certain repairs, which attempt to omit answering a column, will cause these Blk procedures to go into an infinite loop trying to answer the column that was left blank. This whole branch of Blk procedures shouldn't be generated. In a moment, the underlying problem will be discussed. The third procedure output from lesson L_5, 3c-full-P100, passes arguments to the recursive call a little strangely. Whereas both 3c-full and 3c-ful-Blk pass the rightmost unanswered column through the recursion, 3c-full-P100 passes the leftmost column (hence its suffix, P100, which abbreviates "passing the hundreds column"). This only works correctly for three-column problems. Procedure 3c-full-P100 will get stuck if it is given a four-column problem. In fact, it and all its descendents are star procedures due to the strange ways that they answer problems with four or more columns. This branch of the core procedure tree shouldn't be generated. In chapter 10 examines the problem underlying the Blk branch and the P100 branch. The blame is laid on a missing piece of common sense knowledge. The theory should provide some schema for recognizing and building loops as iterations across "similar" objects in a problem state. That is, the knowledge representation language should have, in addition to AND goals and OR goals, a Foreach goal. Such a goal executes its body once for each object in a sequence of objects, e.g., each column in a sequence of columns. The current lack of such a goal forces the learner to build a recursion in order to traverse columns, and this causes the Blk and P100 groups of star bugs.

Lesson L_6 teaches how to solve three column problems of the form NNN−N [SHOW2]. This lesson is a fabrication. At about this point in the Heath textbook, NNN−N problems start appearing in the practice exercises, but there is no specific lesson on the subskill. Lesson L6 has been included in the lesson sequence in order to get around the missing Foreach loop problem. If column traversal were structured around a Foreach loop, then the lesson that teaches how to solve NN−N problems (lesson L_2) would suffice to teach how to do a partial column that occurs anywhere in the problem. Since there is no Foreach loop in the current knowledge representation language, omitting L6 from the lesson sequence means that all the procedures generated from H will be unable to solve NNN−N problems. In particular, all the procedures will manifest one of the two star bugs shown below when they are run through the solver:

*Skip-Interior-Bottom-Blank:

$$\begin{array}{rrr} & & 7\ 9 \\ 3\ 4\ 5 & 3\ 4\ 5 & 8\ 0^{1}7 \\ -\quad\ \ 2 & -\ \ 2\ 2 & -\quad\ \ 9 \\ \hline 3\quad 3\ \times & 3\ 2\ 3\ \checkmark & 7\quad 8\ \times \end{array}$$

*Quit-When-Second-Bottom-Blank:

$$\begin{array}{rrr} & & 7\ 9 \\ 3\ 4\ 5 & 3\ 4\ 5 & 8\ 0^{1}7 \\ -\quad\ \ 2 & -\ \ 2\ 2 & -\quad\ \ 9 \\ \hline 3\ \times & 3\ 2\ 3\ \checkmark & 8\ \times \end{array}$$

(In this and following examples, × marks wrong answers and √ marks correct ones.) In particular, the learner will be unable to generate a correct subtraction procedure. The proper way to avoid these star bugs would be to study the Foreach problem, then make the appropriate changes to the representation language. I haven't done that yet. In the interest of seeing what the theory would generate if that were done, L_6 was added to H. Lesson sequence SF does not have such a lesson. All its predictions involve one of the two star bugs above.

Lesson L_7 teaches how to solve three column problems when one of the columns (but only one) requires borrowing. This lesson refines the fetch pattern that determines where to do the decrement during borrowing. Prior to L_7, the fetch pattern would return both the left-adjacent (tens) column and the leftmost (hundreds) column for borrows that originate in the units column (recall the discussion of the bug Always-Borrow-Left in section 1.1). This lesson modifies the fetch pattern so that only the left-adjacent column is fetched. All the previous lessons have added new subprocedures; lesson L_7 does not. It only modifies existing material. Lesson L_8 is similar. It teaches how to do problems with two adjacent borrows. It does not add a new subprocedure, but only adjusts some fetch patterns. It produces a procedure 3c-2bor (or 3c-2bor-Blk, or 3c-2bor-P100) that can correctly solve any three column problem that does not involve borrowing from zero.

Lesson L_9 teaches how to borrow from zero `[BFZ]`. It produces two procedures that are identical except for their test patterns. The test for 3c-bflz (or 3c-bflz-Blk, etc.) is `{(0 X)}`, which causes the procedure to borrow from zero if the digit to be decremented is a zero. The test pattern for 3c-bflid is `{(ID/ELT X)}`, which makes the procedure borrow across zeros *and ones.* `ID/ELT` is a categorical relation that is defined by the grammar (see figure 2-7) to be true of both kinds of identity elements. Procedure 3c-bflid corresponds to an observed bug, Borrow-Treat-One-As-Zero. The reason the learner produces two procedures is that the lesson is missing a crucial example, one where the digit to be borrowed from is a one (e.g., 514−9). Without this example, the learner can't discriminate which of the two possible test patterns is right. By the way, this illustrates one of the few ways that the grammar has been tailored in order to improve the theory's predictions. If `ID/ELT` were taken out of the grammar, then this bug could not be generated. In general, Sierra's predictions are not particularly sensitive to the grammar. If the grammar works, in that it provides correct parses for all the problem states that the procedures produce, then the model generates about the same set of predictions. The `ID/ELT` case is the exception to this general finding. Anyway, the learner finishes up by taking lesson L_{10}, which teaches how to borrow across multiple zeros. This lesson has no effect on 3c-bflz, since the procedure can already do that.

The core procedure tree has two 2-way branches and one 3-way branch. It could have as many as 2x2x3=12 final procedures. In fact, there are just 2. The other branches are pruned when the learner is unable to assimilate the next lesson. For instance, the branch for $T \leq B$ as the test for borrowing (i.e., procedure 2c-1bor-leq) is terminated at lesson L_5 because one of the examples in that lesson is

$$\begin{array}{r} 9\ 8\ 5 \\ -\ 6\ 2\ 5 \\ \hline 3\ 6\ 0 \end{array}$$

The procedure expects the units column to have a borrow, but the example does not have a borrow there. The learner could install a new subpocedure that would avoid borrowing whenever $T=B$. However, lesson L_5 is already introducing a new subprocedure. The learner cannot introduce two subprocedures in one lesson because that would violate the one-disjunct-per-lesson felicity condition. So this branch of the core procedure tree is pruned. Intuitively, such pruning represents remediation.

It might seem that the model is doing a very poor job of explaining where student's bugs come from. It seems to explain only two bugs, N−N-Causes-Borrow and Borrow-Treat-One-As-Zero. This is no great feat. Any inductive account of learning could explain these two bugs since their "causes" lie in the absence of certain crucial training examples. However, the real test of the learner is not what bugs it produces *directly*, but what structures it assigns to the procedures that it produces. A procedure's structure has a direct affect on deletion and local problem solving. By examining the bugs produced by the solver, one can ascertain whether the procedure's structures are plausible or not.

2.9 The solver

Each of the procedures output by the learner is given, one by one, to the solver. The solver "takes" a diagnostic test by applying the procedure to solve each problem on the test. The solver has two parts, called the *interpreter* and the *local problem solver.* The interpreter executes the procedure. The local problem solver executes repairs whenever the interpreter's execution is halted by an impasse. To describe this in more detail: When the interpreter reaches an impasse, the local problem solver selects one of a set of repairs. Applying the selected repair changes the internal state of the interpreter in such a way that when the interpreter resumes, it will no longer be stuck. The local problem solver may (or may not) create a patch, which will cause the same repair to be chosen if ever that impasse occurs again. Stable bugs are accounted for by creating and retaining patches for long periods; bug migrations result from short-term patch retention. By systematically varying the choice of repairs and the use of patches during repeated traversals of the test, the set of all predictions that can be generated from the given procedure can be collected.

This section describes how the interpreter and the local problem solver work. For illustrations, it uses the procedure whose AOG is sketched in figure 2–12a. The procedure is called 3c-2bor in figure 2–14. It doesn't "know" how to borrow from zero. It can solve problems like *a* or *b*, but not ones like *c*:

```
a.    2 3      b.    4 5 1      c.    5 0 7
    - 1 7          -   8 7          -   2 8
```

This section is constructed as a scenario that traces the execution of the AOG on problem c. The AOG reaches an impasse when it tries to borrow from the zero. Local problem solving repairs the interpreter's state, allowing the interpreter to finish the problem. Depending on the repair selected by the local problem solver, various bugs are generated.

By convention, AOG's are started by calling their root goal on the initial problem state. In this case, START is called with the whole subtraction problem as its argument. START doesn't do anything interesting. It merely calls SUB (if addition were part of this procedure's competence, START would have to decide whether to call SUB or ADD). SUB's purpose is to find the units column and pass it to 1/SUB. 1/SUB's job is to decide whether the given problem is a regrouping problem, a single-column subtraction problem or a multi-column subtraction problem. It finds that there are several columns to be subtracted, so MULTI is called with the units column as its argument.

MULTI is an AND goal that implements a loop across the columns of the subtraction problem. AND goals execute their rules in left-to-right order. MULTI executes its first rule, which calls SUB1COL and passes the units column as its argument. SUB1COL is an OR goal that chooses a method for answering its column. OR rules are tested in left-to-right order. SUB1COL's first rule tests for a blank in the bottom of the current column, which is the units column of 507–28. The bottom of the column is 8, so the test fails. SUB1COL's second rule tests whether the top digit of the column is less than the bottom digit. Since $7<8$, the second rule is executed, and BORROW is called. BORROW is an AND goal, whose definition is:

```
BORROW (T B A)   Type: AND
     1.  {} --->  (1/BORROW T)
     2.  {} --->  (2/BORROW T B A)
```

As the first rule is executed, **1/BORROW** is passed the value of **BORROW**'s first argument, which happens to be the parse node for the top digit of the units column (node 31 in figure 2–8). The usual call-by-value semantics applies for argument passing. **1/BORROW** gets T's value rather than its intension (e.g., lazy evaluation, or call-by-name). The issue of passing intensions versus extensions is a complex one, which is discussed in appendix 8.

1/BORROW is a trivial OR goal that calls REGROUP. REGROUP's purpose is to "regroup" a ten to become ten ones. REGROUP is an AND goal that first calls BORROW/FROM and passes it the top digit-place in the column that is left adjacent to REGROUP's argument. BORROW/FROM is a trivial OR goal, whose definition is:

```
BORROW/FROM (TD)   Type: OR
    1.  {} --->   (OVRWRT TD (Sub1 (Read TD)))
```

BORROW/FROM's argument, TD, is currently bound to the parse node for the top digit of the tens column (node 27 in figure 2–8). Since the rule's pattern is null, it always matches. The action, like all rule actions, is an evaluable form, in the Lisp sense. It will attempt to call the subgoal OVRWRT, passing it the value of TD and a number. The number will be calculated by the function nest `(Sub1 (Read TD))`. In this case, the function nest tries to subtract one from the top digit of the tens column, which is zero (the problem is $507-28$). Trying to decrement zero violates a precondition of `Sub1`. Violating a precondition causes an impasse. Local problem solving is initiated to find a way to change the state of the interpreter in order to make it continue.

There are five kinds of impasses. Precondition violations, such as the one just discussed, are one kind. For completeness, the other four are listed below, but will not be discussed further:

1. Halt: OR rules may only run rules that have (a) not been run before in attempting to satisfy this invocation of the goal, and (b) have true test patterns. If there are no such rules, then a halt impasse occurs. The interpreter can't decide which rule to run, so it invokes local problem solving.

2. Ambiguity: The fetch patterns on AND rules are used to bind certain pattern variables (nicknamed "output" variables) whose values are then used in the rule's action. If a fetch pattern matches ambiguously, so that there are two or more values for an output variable, then an ambiguity impasse occurs. The interpreter can't decide which way to match the fetch pattern, so it invokes local problem solving.

3. Infinite loop: If the interpreter detects an infinite loop (e.g., because the goal stack depth exceeds some very large threshold), then an infinite loop impasse occurs.

4. Crazy notation: If the parser is unable to parse the current problem state, which means that it is not syntactically well-formed with respect to the grammar, then an impasse occurs.

Returning to the scenario, figure 2–15 shows the interpreter's state as local problem solving begins. The interpreter's state consists of a *goal stack*, and a mode bit called *microstate*. Microstate indicates whether the interpreter is calling (microstate = Push), or returning (microstate = Pop). The format of the interpreter's state is important because the interpreter's state is where the local problem solver does its problem solving. The general idea of local problem solving is that the local problem solver can do anything it wants to the interpreter's state as long as it leaves the state set so that the interpreter will continue. The local problem solver can not change the problem state (i.e., write symbols on the page), it can only change the interpreter's state.

```
Microstate = Push


(BORROW/FROM (CELL 27))
(REGROUP (CELL 31)
(1/BORROW (CELL 31)
(BORROW (CELL 31)(DIGIT 12)(BLK 34))
(SUB1COL (CELL 31)(DIGIT 12)(BLK 34))
(MULTI (CELL 31)(DIGIT 12)(BLK 34))
(1/SUB (CELL 31)(DIGIT 12)(BLK 34))
(SUB (PROBLEM 53)
(START (PROBLEM 53))
```

Figure 2–15
The interpreter's state—microstate and the goal stack—at the time of the impasse.
Goal arguments are shown as the main category and the serial number of the parse node (see fig. 2–8).

Which changes the local problem solver chooses is left open to individual variation so that the model will capture the fact that different subjects repair the same impasse different ways. (Indeed, the same subject may even repair the same impasse different ways at different times, an account for bug migration.) However, unrestricted changes to the interpreter's state gives the local problem solver a great deal of power. It could, for example, run the AOG in some kind of simulation mode. The model would thus be able to generate just about anything by hypothesizing the appropriate local problem solving. In short, a tricky problem of repair theory is to constrain the local problem solver is such a way that the theory is refutable, but still empirically successful. The current version of repair theory postulates five operators, called repairs, that modify the interpreter state:

Noop	pops the stack. When the interpreter resumes, it will think the top goal has been accomplished. Essentially, this repair makes the interpreter skip the stuck goal, turning it into a null operation, or "no op" in computer jargon.
Backup	pops the stack to the highest (most recently called) OR then sets microstate to Push. This will cause the interpreter to choose a different OR rule to call. Put intuitively, the "student" decides to back up to the last place that a choice was made in order to go the other way instead.
Quit	pops the stack back to the root goal of the AOG, then sets microstate to Pop. Intuitively, the "student" decides to give up on this problem and go on to the next test item.
Refocus	resets the arguments of the top goal in such a way that the precondition is no longer violated. It does so by rematching the most recently used fetch pattern. This causes the interpreter to execute the top goal with different arguments, "shifting its focus of attention" to avoid the impasse. (Figure 2–16 shows Refocus applied two different ways to the impasse currently under discussion.)
Force	has different affects depending on the kind of impasse it repairs. If the impasse is a halt impasse, where none of the rules have true test patterns, then the Force repair will pick one of the rules and cause the interpreter to execute it. If the impasse is an ambiguity impasse, where a fetch pattern matches several ways so that it is ambiguous which values to pass as subgoal arguments, then the Force repair will pick one of the possible matches and cause the interpreter to use it.

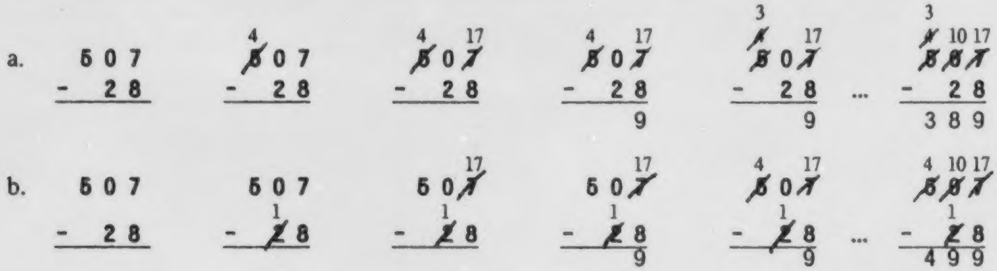


Figure 2–16

(a) Refocus relaxes a relation that says that the column to borrow from should be adjacent to the column borrowed into. This causes the decrement to be placed in the hundreds column. This application of the repair generates a very common bug (42 occurrences in the Southbay data) called Borrow-Across-Zero.

(b) Refocus relaxes a relation that says that the cell to borrow from should be the first (top) cell in the column. This application of the repair generates a rare bug (1 occurrence in the Southbay data) called Borrow-From-Bottom-Instead-of-Zero.

Given these repairs, local problem solving is simple; it is just selection and application of a repair. However, this simple regime can generate quite complex bugs. Repairs often cause secondary impasses. Since they don't actually fix the underlying defect in the student procedure but rather just get the interpreter running again in the most expedient way, they often leave the problem in a state that will cause further impasses. Repairing those impasses may lead to tertiary impasses. In principle, there could be an arbitrarily long causal chain. In practice, one rarely sees chains longer than six impasse-repair occurrences.

The above description of the local problem solver is a little simplified. There are a few complications concerning the creation and use of patches. A *patch* is an association of a repair and impasse that the local problem solver creates in order to cache (save) the results of a particular occurrence of local problem solving. Another complication whose discussion will be put off for later concerns *critics*, which block the selection of repairs under certain circumstances.

To return to the scenario, suppose that the local problem solver chooses the Noop repair. This causes the `BORROW/FROM` goal to return to `REGROUP`, having made no changes in the initial problem state. `REGROUP`, which is an AND goal, goes on and executes its second rule which calls `BORROW/INTO` passing it the parse node for the top digit in the units column. `BORROW/INTO`'s definition follows:

```
BORROW/INTO (TD)  Type: OR
     1.  {} ---> (OVRWRT TD (Concat (One) (Read TD)))
```

This OR goal merely "adds" ten to the given digit by concatenating a one to its left, and has `OVRWRT` write the "sum" over the digit. In this case, calling `BORROW/INTO` yields the problem state in figure 2–17b. Control returns to `BORROW`, popping `BORROW/INTO`, `REGROUP` and `1/BORROW` on the way. `BORROW` executes its last rule, which takes the column difference for the units column. Now the problem appears as in figure 2–17c. `BORROW` is popped, and control returns to `MULTI`. The procedure is done with the units column. It still has the tens column and the hundreds column left to do. These are processed uneventfully, with no impasses, so it is best to stop the scenario here. Figure 2–17 shows the remaining problem states between here and the end of the problem's solution.

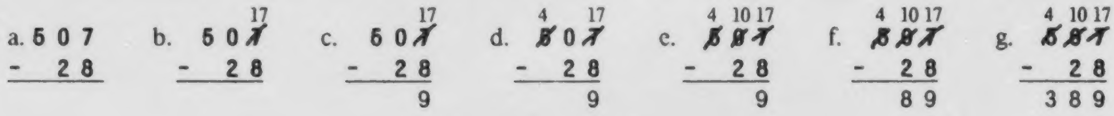


Figure 2–17
Sequence of problem states, omitting crossing out actions, for Stops-Borrow-At-Zero

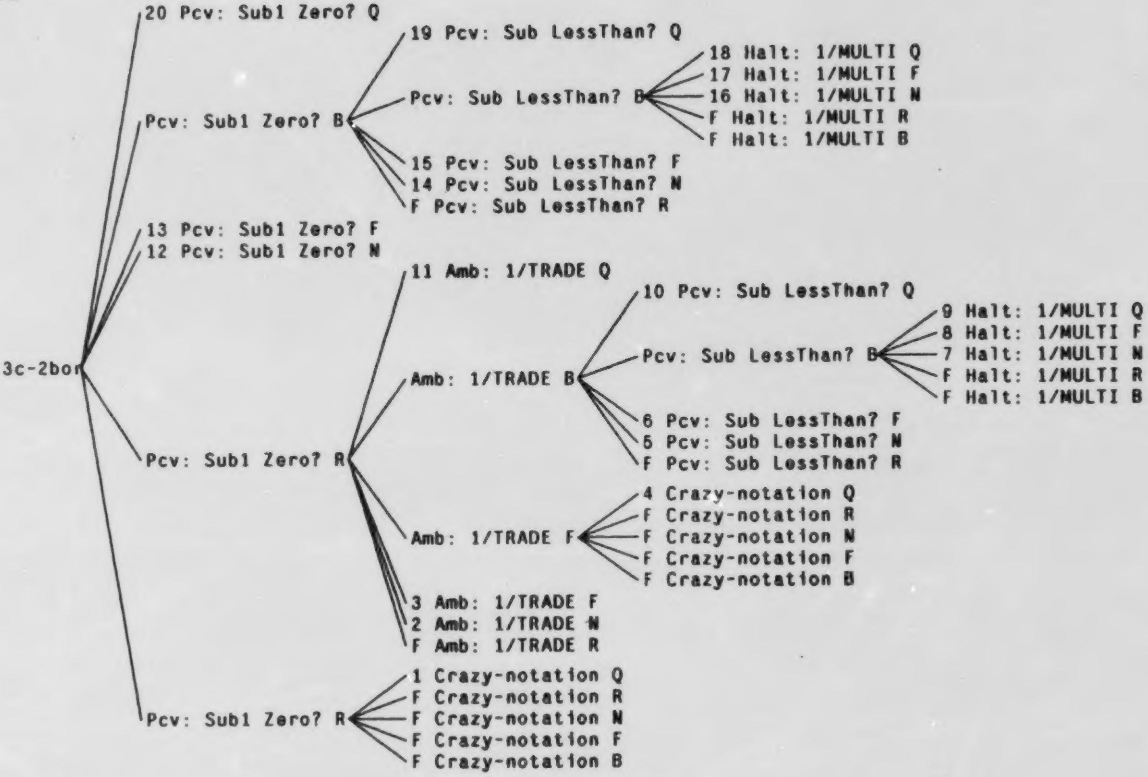
This solution to the subtraction problem involved taking the Noop repair to the impasse. The solution is characteristic of a common bug (64 occurrences in the Southbay data) called Stops-Borrow-At-Zero. Taking other repairs would produce other solutions to the exercise, some of which will be bugs. Figure 2–16 illustrates two bugs generated by taking the Refocus repair instead of the Noop repair on this exercise.

When Sierra is given a diagnostic test and a procedure, it will generate solved tests corresponding to all possible combinations of repairs to the impasses it encounters. This varying of repairs to impasses is a prolific source of predictions. Figure 2–18 displays this by sketching the impasse-repair tree for this procedure when it "takes" the diagnostic test shown in figure 2–1. The solver reaches its first impasse on the test's 14th problem, 102 − 39, because the problem requires borrowing from a zero. Each of the leftmost branches in the tree corresponds to a different way to repair this impasse. The six nodes are labelled with the impasse: "Pvc: Sub1 Zero?" identifies it as a precondition violation impasse where `Sub1`'s error test, `Zero?`, is true when `Sub1` was called. The letter following the impasse identification is a code for the repair that was applied: Q for Quit, B for Backup, F for Force, N for Noop and R for Refocus. There are always at least five branches for each impasse because there are five repairs. There may be more, because some repairs can apply more than one way. Notice that there are two nodes labelled with R among the first six branches. These correspond to two ways to apply the Refocus repair. If a repair is not applicable to a certain impasse or the repair fails to fix the impasse when it is applied, then the corresponding node has "F," for filtered, as a prefix.

Some of the repairs lead to further impasses. When this happens, the node has a subtree to its right (e.g., the Backup repair to this impasse). On the other hand, if the test can be completed without further local problem solving, the node is a leaf of the tree and it has a number as its prefix. The number is an index into the table beneath the tree. For instance, solved test 1 has exactly the answers produced by the bug Borrow-Won't-Recurse. Solved test 2 generates exactly the answers produced by a set of three bugs. (Actually, the "bugs" with exclamation points in their names are called "coercions." See appendix 2.)

Sierra's solver has a switch that controls whether it will generate bug migrations or not. The above impasse-repair tree was generated by turning off bug migration. This causes the solver to use patches so that whenever an impasse occurs that has occurred before on the test, the solver will apply the same repair that it chose before. If bug migration were left on, then the solver would generate a huge number of solved tests. Essentially, each occurrence of the original impasse (i.e., the borrow-from-zero impasse in this case) would yield an impasse-repair tree. There are seven borrow-from-zero columns on the diagnostic test. Hence, Sierra would generate approximately 20^7 solved tests if bug migration were left on. Most of these would be identical, probably, but still Sierra would have to generate them all, if observational adequacy were to be thoroughly assessed. Needless to say, this is not what is done. For practical reasons, observational adequacy is assessed only with respect to bugs, not bug migrations.

A


20 Pcv: Sub1 Zero? Q
19 Pcv: Sub LessThan? Q
18 Halt: 1/MULTI Q
17 Halt: 1/MULTI F
Pcv: Sub LessThan? B
16 Halt: 1/MULTI N
Pcv: Sub1 Zero? B
F Halt: 1/MULTI R
F Halt: 1/MULTI B
15 Pcv: Sub LessThan? F
14 Pcv: Sub LessThan? N
F Pcv: Sub LessThan? R
13 Pcv: Sub1 Zero? F
12 Pcv: Sub1 Zero? N
11 Amb: 1/TRADE Q
10 Pcv: Sub LessThan? Q
9 Halt: 1/MULTI Q
8 Halt: 1/MULTI F
3c-2bor
Pcv: Sub LessThan? B
7 Halt: 1/MULTI N
Amb: 1/TRADE B
F Halt: 1/MULTI R
F Halt: 1/MULTI B
6 Pcv: Sub LessThan? F
5 Pcv: Sub LessThan? N
Pcv: Sub1 Zero? R
F Pcv: Sub LessThan? R
4 Crazy-notation Q
F Crazy-notation R
Amb: 1/TRADE F
F Crazy-notation N
F Crazy-notation F
F Crazy-notation B
3 Amb: 1/TRADE F
2 Amb: 1/TRADE N
F Amb: 1/TRADE R
1 Crazy-notation Q
F Crazy-notation R
Pcv: Sub1 Zero? R
F Crazy-notation N
F Crazy-notation F
F Crazy-notation B

B

1. {Borrow-Won't-Recurse}
2. {Borrow-Across-Zero !Touched-Zero-Is-Ten !Touched-Double-Zero-Is-Quit}
3. {Borrow-Across-Zero !Touched-Double-Zero-Is-Quit}
4. {Borrow-Across-Zero !Touched-0-Is-Quit}
5. {Borrow-Across-Zero !Touched-0-N = Blank !Touched-Double-Zero-Is-Quit}
6. {Borrow-Across-Zero !Touched-0 – N = N !Touched-Double-Zero-Is-Quit}
7. {Borrow-Across-Zero !Touched-0-N = Blank !Touched-Double-Zero-Is-Quit}
8. {Borrow-Across-Zero !Touched-0 – N = 0 !Touched-Double-Zero-Is-Quit}
9. {Borrow-Across-Zero !Touched-0-Is-Quit}
10. {Borrow-Across-Zero !Touched-0-Is-Quit}
11. {Borrow-Across-Zero !Touched-0-Is-Quit}
12. {Stops-Borrow-At-Zero}
13. {Borrow-Add-Decrement-Insteadof-Zero}
14. {Blank-Insteadof-Borrow-From-Zero}
15. {Smaller-From-Larger-Insteadof-Borrow-From-Zero}
16. {Blank-Insteadof-Borrow-From-Zero}
17. {Top-Insteadof-Borrow-From-Zero}
18. {Borrow-Won't-Recurse}
19. {Borrow-Won't-Recurse}
20. {Borrow-Won't-Recurse}

Figure 2–18
(a) Impasse-repair tree for the core procedure 3c-2bor.
(b) Debuggy's analysis of each of the 20 predicted test solutions.

2.10 Observational adequacy

In principle, the following simple procedure is used to test the observational adequacy of the theory:

1. Administer diagnostic tests to a large number of students.
2. Collect the test sheets and code the answers into machine-readable form.
3. Analyze each test solution with Debuggy, thus redescribing it as a set of bugs.
4. Call the set of all test solutions (represented as bug sets) OST — the observed test solutions.
5. Formalize the textbooks used by the students, producing several lesson sequences.
6. Formalize an initial state of knowledge, KS_0.
7. Run Sierra's learner over each lesson sequence. This produces one core procedure tree per lesson sequence.
8. Formalize a diagnostic test form.
9. For each procedure in each core procedure tree, run Sierra's solver over the diagnostic test. This produces one impasse-repair tree per core procedure.
10. For each leaf of each impasse-repair tree (except the leaves representing filtered repairs), analyze the leaf's solved test with Debuggy. This produces one bug set per solved test.
11. Call the set of all such bug sets PST — the predicted solved tests.
12. Calculate $OST \cap PST$, $OST - PST$ and $PST - OST$.
13. Separate the star bugs, if any, from $PST - OST$.

In practice, things are more complex. Steps 5 and 6 involve tailoring. Trying different lesson sequences led to the discovery that omitting the regrouping lesson causes the model to generate several new, valid predictions. In principle, the initial knowledge state, KS_0, should be a rich source of variation since it is likely that not all students have the same initial understanding. In the Southbay experiment, just one KS_0 was used. Two others were tried, briefly, but they produced almost the same observation adequacy as the chosen KS_0*.

The steps that involve Debuggy, steps 3 and 10, are actually quite a bit more complex than described so far. In fact, most of the rest of this section will be concerned with the practical aspects of using bug-based observational adequacy. First, the reality of step 3, analyzing the observed test solutions, will be briefly described (VanLehn, 1981, covers it in detail). Then the reality of step 10, analyzing the test solutions generated by Sierra, will be described. Finally, the Southbay numbers for $OST \cap PST$, $OST - PST$ and $PST - OST$ will be presented.

---

* The observed bug Borrow-From-Bottom-Insteadof-Zero can be generated by modifying the grammar rule that defines COL. The observed bug Zero-Insteadof-Borrow can be generated by modifying the primitive Sub function so that it implements $\max(0, x-y)$ instead of $|x-y|$. These are the only observed bugs that are generated by non-standard KS_0 (that I know of) and not by the standard KS_0.

Analyzing the Southbay data

Not all students have bugs. Some students know the correct algorithm. Others migrate among several bugs during the test. On one experiment, the students fell into five categories in roughly the following proportions:

7	(10%)	Perfect score. No errors of any kind.
61	(46%)	Knows the correct algorithm; errors due to slips alone.
34	(26%)	Has a bug or a set of bugs (plus perhaps some slips as well).
14	(10%)	Intra-test bug migration (plus perhaps some bugs and slips).
11	(8%)	Errors cannot be analyzed.
134	(100%)	total

These proportions vary with the grade level. The above proportions are for third graders tested late in the year. In general, the older the student population, the greater the proportion of students in the slips category and the smaller the proportion in the bugs and bug-migration categories. In the early third grade, for example, students in the bugs categories constitute over 50% of the sample instead of 26%. This shift is not surprising. In the early grades, the students have not yet been taught the whole algorithm. When given the diagnostic test, they will have to do local problem solving to answer most of its items. Hence, younger students are more likely to have bugs, and older students are more likely to have only slips.

The figures just given come from a special experiment where students were given two tests on consecutive days with no intervening instruction (this experiment is called the Short-term study in VanLehn, 1981). For each problem on one test, there was a very similar problem on the other test. These matched-item tests were designed to provided enough redundancy that cases of bug migration could be found. The usual twenty-item test is too short for one to have confidence in bug migration analyses.

The two-test experiment allowed assessment of the short-term stability of students' errors. Various kinds of errors are expected to have differing kinds of short-term stability. Slips are expected to vary widely over two tests given a short time apart. There may be no slips on one test and several on another. If there are slips on both tests, they are not expected to occur on the same problems. Impasses on the other hand are expected to remain in evidence across tests. Because impasses derive from the student's core procedure, and it is assumed that core procedures are stable in the short term, impasses should be stable in the short term as well. An impasse may show up differently on the two tests. It might manifest as a bug on one test and as a different bug or as intra-test bug migration on the next test. What would be unexplained is a impasse that is present on one test but absent on the other. These considerations prompt the following tabular summary of the percentage of students exhibiting the various kinds of stability:

3	(4%)	No errors on either test.
32	(49%)	Stable correct procedure; changes due to slips alone.
3	(4%)	Stable bugs; changes due to slips alone.
12	(18%)	Stable impasses; changes due to repairs (often along with slips and stable bugs)
13	(19%)	Appearing and disappearing impasses (with slips and stable bugs).
4	(6%)	Errors cannot be analyzed.
67	(100%)	total

The stability patterns of the students in the first four categories (75%) conform to expectations, while the behavior of the students in the remaining two categories (25%) remain unexplained.

These stability data show that the older view of errors as due to either bugs (deterministic, repeatable errors) or slips (underdetermined, stochastic errors) is incomplete. On that view, bugs were stable and only slips could account for short-term instabilities. The impasses/repair notions contribute substantially to our ability to understand short-term instabilities (in addition to their role as an explanation of the acquisition of bugs).

However, a significant proportion of the tests (8% of the static, one-test data, and 25% of the stability data) cannot yet be analyzed even with these advances. Most of these students are in the unanalyzable category because the tests were simply not long enough to give the analysts a large enough sample of their behavior. Without a large and variegated set of errors, it is sometimes impossible to disambiguate the various possible explanations for the student's errors. Such ambiguous analyses are counted in the category unanalyzable. In other cases, species of behavior that have not yet been formalized were apparent. Some students appeared to "punt" the test by struggling through the first part of it, then giving up and using some easily executed buggy procedure. There seemed to be several cases of cheating by looking at a neighbor's paper. In short, there will undoubtedly be some errors that have rather uninteresting causes and hence can properly be left unanalyzed. My belief is that we have not quite reached that level of understanding yet. There probably remain some undiscovered, interesting mechanisms that may further our understanding of errors as much as the impasse/repair process did.

The figures given above were derived by hand analysis of the matched-test data. This is necessary because Debuggy cannot analyze bug migrations. It can only find bugs that are stable across the whole test. Its analyses of the same data, and the Southbay data, are shown below:

	Southbay		Short-term	
No errors	98	(10%)	14	(10%)
Errors due to slips alone	198	(20%)	41	(31%)
Has a set of bugs	340	(34%)	35	(26%)
Errors cannot be analyzed	377	(36%)	44	(33%)
total	1013	(100%)	134	(100%)

Notice that about a third of the students could not be analyzed by Debuggy. The main reason that so many students could not be diagnosed by Debuggy is that they were making too many slips (VanLehn, 1981, discusses this issue in detail). However, there was also some bug migration among the unanalyzable students, as well as a non-trivial amount of truly puzzling behavior. For Debuggy to do better, it would have to have more redundant test items. Then it could locate slips in the way that the human analysts did, by comparing a student's performance on identical or nearly identical problems. On the other hand, Debuggy is totally objective. Unlike me, it does not "hope" for the occurrence of certain bugs that would confirm the predictions of the theory. In service of objectivity, its opinions are used throughout this document as the definition of "bug occurrence." Also, subsequent references to the Southbay data will include the Short-term data as well.

Debuggy cannot invent new bugs. Its inventiveness is limited to creating new *sets* of bugs. Debuggy has a database of bugs that it combines to form analyses. (Creating a new set of bugs may sound trivial, but it is actually quite difficult since many bugs interact with each other in complex ways. See Burton, 1982.) The method used to discover new bugs for the database is to use Debuggy as a filter to remove students whose behavior is adequately characterized by existing bugs. This leaves the human analysts to concentrate on discovering any systematicity that lurks in what Debuggy considers unsystematic behavior. When even the barest hint of a new bug is uncovered by the experts, it is formalized and incorporated in Debuggy's database. That way, Debuggy will discover any subsequent occurrences of the bug, even when it occurs with other bugs, and even when it interacts in non-linear, complex ways with those bugs. At the end of the

Southbay experiment, the database had grown to 103 bugs. The tests had been thoroughly examined by myself and two other analysts. We were confident that few bugs, if any, lurked undiscovered in the data. As will be seen shortly, our confidence was misplaced. Sierra invented some bugs that we did not think of, and six of them turned out to be observed bugs!

Generating and analyzing the predicted test solutions

To generate the Southbay predictions, Sierra's learner traversed lesson sequences H, SF and HB, producing three core procedure trees. In principle, all core procedures in all three core procedure trees would be submitted individually to Sierra's solver, along with the appropriate deletion-generated core procedures. This would be 63 core procedures, for the trees just given. However, many of the core procedures are quite similar. Others are known to be "star" core procedures in that all the solved tests that they will generate will be marred by star bugs (e.g., the P100 branch of H's tree is a "star" branch. As discussed in section 2.8, it would be blocked by adding a "Foreach" loop to the knowledge representation language). Running these redundant and/or star core procedures through the solver would only generate more instances of already generable bugs, not bugs that could not be generated some other way. Consequently, a subset of the 63 core procedures was selected and run through the solver*. Thirty core procedures were submitted to the Sierra's solver, generating 30 impasse-repair trees. The trees' leaves yielded 893 solved tests. The solved tests were analyzed by Debuggy.

The analysis of predicted test solutions has to be more stringent than analysis of observed test solutions. Basically, a predicted test solution counts as analyzed only if Debuggy's bug set for it *exactly matches* its answers. Inexact analyses doesn't make sense, since Sierra does not make slips nor did it do bug-migration (bug migration is turned off in the solver when generating the impasse-repair trees). However, exact matching turned out to be too stringent a criterion. In Debuggy's versions of certain bugs, there is special code inserted to handle rare cases, such as the bug running off the left edge of the problem while borrowing. In Sierra, such cases are handled automatically by the usual local problem solving mechanism. However, the special case code in Debuggy's bugs occasionally would not correspond to any of the various impasse-repair combinations that Sierra generated. The net effect is that none of Sierra's solved tests would exactly match the Debuggy's bug's performance. In almost all cases, the analysis was off by one problem. That is, Debuggy's analysis would match 19 out of 20 answers on a solved test, but the 20th problem's answer would not match exactly (although the rightmost few digits would often be the same). Consequently, the analyses were divided into two classes: perfect and almost perfect. The latter class is the off-by-one analyses.

---

* The subset included all the core procedures from the H core procedure tree, except the P100 branch and the Blk branch. From the SF core procedure tree, only the procedures that are in a direct line from the root to the "ok" procedure were run. From the HB core procedure tree, only three procedures were run: 3c-1bor, 3c-1bfz and 3c-1bfid. All the deletion-generated procedures from H and HB were run, except for those that are generated before three-column subtraction is taught. Since a Foreach loop would change the early procedures' structures, the procedures that would be generated from them by deletion would be different as well.

	Good Proc's		No Loop Proc's		All Proc's	
No errors	2	(1%)	0	(0%)	2	(0%)
Perfect	209	(83%)	289	(46%)	498	(56%)
Almost perfect	28	(11%)	91	(14%)	119	(13%)
Analyzable	4	(2%)	145	(23%)	149	(17%)
Unanalyzable	8	(3%)	111	(17%)	125	(14%)
total	251	(100%)	642	(100%)	893	(100%)

Figure 2-19

The 893 solved tests generated by Sierra, categorized by how well Debuggy's analysis matched.

At first, Debuggy's database of bugs was insufficient to analyze very many of the solved tests. Only 1% of the tests could be perfectly or almost perfectly analyzed. To solve this problem, Debuggy's database was expanded from 103 bugs to 147 bugs. The 44 added bugs included star bugs as well as bugs that could plausibly be observed bugs. Any bug was added that would get more of the solved tests analyzed. However, a point of diminishing returns was reached. Figure 2-19 shows the number of solved tests at the point where I stopped adding bugs to the database. The solved tests are separated into two groups corresponding to two groups of core procedures. The "Good" group (the left column of the figure) contains solved tests from core procedures that "know" how to loop across columns. These would presumably be roughly the same when the "Foreach" loop problem is fixed. The other group (middle column) comes from core procedure that suffer the effects of not being able to process multi-column problems. I tended to add few bugs to the data base in the service of their analysis since I expect that they will not be with us much longer. The figures reflect this. Enough bugs were added to the data base to analyze 95% of the "Good" solved tests, but only 60% of the solved tested were analyzable from the other set of solved tests.

After the 44 bugs were added to Debuggy's database to Sierra, the Southbay data was reanalyzed. Six of the 45 bugs turned up in the analyses*. Two of them even occurred rather frequently — seven times each. This was quite a surprise.

Results

When Debuggy analyzed the 1147 observed test solutions that constitute the Southbay data, it found bug sets for 375 of them. However, many of the solved tests received the same bug set. The eleven most common analyses are listed in figure 2-20. One can see that the frequency of occurrence falls off rapidly. There are 134 distinct bug sets. Most of them (99) occur only once. Appendix 2 lists all the observed bug sets. Debuggy found bug sets for 617 of the predicted test solutions (including almost perfect as well as perfect matches). There were 119 distinct bug sets. Appendix 4 lists them. With 134 bug sets in OST and 119 bug sets in PST, the observational adequacy can (at last!) be calculated:

---

* The bugs are: Borrow-Across-Second-Zero (7 occurrences), Doesn't-Borrow-Except-Last (1 occurrence), Only-Do-Units (1 occurrence), Smaller-From-Larger-Except-Last (3 occurrences), Smaller-From-Larger-Insteadof-Borrow-Unless-Bottom-Smaller (7 occurrences), and Top-Instead-Of-Borrow-From-Zero (1 occurrence).

OST∩PST	11 bug sets
OST−PST	113 bug sets
PST−OST	118 bug sets, of which 53 contain star bugs.

Appendices 5 and 6 list the bug sets in each of OST∩PST, OST−PST, and PST−OST. Interestingly, the bug sets in OST∩PST happen to include several of the most common observed bug sets. Of the eleven most common bug sets (see figure 2–20), it includes bug sets 2, 3, 4, 5 and 8.

Because a bug-based measure of observational adequacy is being used, rather than one based on raw test solutions, the figures above can be dissected to discover why the observation adequacy is the way it is. For instance, why is the most common bug set, {Smaller-From-Larger}, not in OST∩PST? To answer such questions, each bug set in OST−PST is intersected with the bug sets in PST−OST. This forms four new categories, depending on whether the intersection is empty or not:

	non-empty	empty
OST−PST	76	47
PST−OST	68	40

This chart shows that 76 bug sets from OST−PST had at least one bug generated by the model. These 76 bug sets include the popular bug set {Smaller-From-Larger} because it overlapped with the predicted bug set {Smaller-From-Larger *Only-Do-First&Last-Columns}. The second bug in the predicted bug set is a star bug. It is generated because there is no Foreach loop in the representation language. When a Foreach loop is added, {Smaller-From-Larger} will be properly predicted. Using overlaps, one can find out what needs to be done to improve observational adequacy. The overlaps are listed in appendices 5 and 6.

From the overlaps, one can see that the main reason that OST∩PST is so small is that many students have bugs *in addition* to the bugs generated by the model. For instance, there is a set of bugs that the model does not generate that all involve mis-answering columns whose top digit is a zero. The bugs in this class are (appendix 1 contains descriptions of these bugs):

	occurs	bug
1.	103	(Smaller-From-Larger)
2.	34	(Stops-Borrow-At-Zero)
3.	13	(Borrow-Across-Zero)
4.	10	(Borrow-From-Zero)
5.	10	(Borrow-No-Decrement)
6.	7	(Stops-Borrow-At-Zero Diff-0-N=N)
7.	6	(Always-Borrow-Left)
8.	6	(Borrow-Across-Zero !Touched-0-Is-Ten)
9.	6	(Borrow-Across-Zero Diff-0-N=N)
10.	6	(Borrow-Across-Zero-Over-Zero Borrow-Across-Zero-Over-Blank)
11.	6	(Stops-Borrow-At-Zero Borrow-Once-Then-Smaller-From-Larger Diff-0-N=N)

Figure 2–20
The eleven most common bug sets, with the number of times each occurred

Diff-0 − N = 0
Diff-0 − N = N
0 − N = N-After-Borrow
0 − N = 0-After-Borrow
0 − N = N-Except-After-Borrow
0 − N = 0-Except-After-Borrow

Suppose the model were able to generate these bugs in such a way that they could occur in any bug set that the model currently generates. This is not so implausible; a story for how that might have happened will be presented. If any of the above 0 − N bugs could occur in the bug sets of PST, then OST∩PST would *triple* in size, becoming 43 bug sets large. The point is that small increases in the productivity of the model with respect to primitive bugs can translate into big gains when counting bug sets. This effect can be seen clearly with the aid of a toy example. Suppose that there were only ten primitive observed bugs, and that the model generates just two of them. Suppose further that that all 45 pairs of these ten bugs occur as bug sets. Only one of the observed bug sets will be in OST∩PST. If the model generated three or four bugs instead of two, the figures would change, but not rapidly:

2 bugs	3 bugs	4 bugs	
1	3	6	OST∩PST
16	21	24	OST − PST with non-empty intersections
28	21	15	OST − PST with empty intersections

If the model is generating less than half of the observed primitive bugs, then counting bug sets makes it look much worse. (Conversely, if it generates more than half the primitive bugs, counting bug sets makes it look much better.) This suggests measuring observational adequacy with respect to primitive bugs, rather than bug sets.

Let OB be the union over all the bug sets in OST. OB is a set of 76 bugs. Let PB be the union over PST. PB contains 49 bugs. Then:

OB∩PB	25 bugs
OB − PB	51 bugs
PB − OB	24 bugs, of which 7 are star bugs.

Appendix 3 lists these bugs. Figure 2-21 displays the figures as a Venn diagram. Essentially, these figures say that half of the theory's predictions are confirmed. On the other hand, there is much work left to do, because two-thirds of the observed bugs are not yet accounted for.

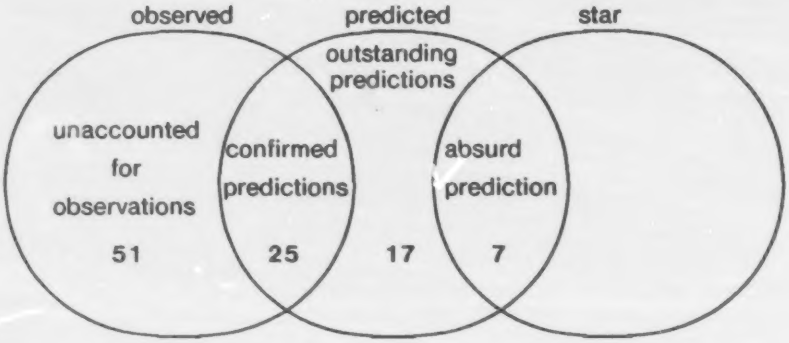


Figure 2-21
Venn diagram showing relationships and sizes of the sets of predicted, observed and star bugs.

2.11 A comparison with other generative theories of bugs

The numbers presented here are difficult to understand without some point of reference. Two such points are provided by earlier generative theories of subtraction bugs. An early version of repair theory is documented in Brown and VanLehn (1980). Its empirical adequacy can be compared with the present theory's. Clearly, this theory will do better since it includes the ideas of its predecessor. Another generative theory of subtraction bugs was developed by Richard Young and Tim O'Shea (1981). They constructed a production system for subtraction such that deleting certain of its rules (or adding certain other rules, in some cases) would generate observed bugs. They showed that these mutations of the production system could generate many of the bugs described in the original Buggy report (Brown & Burton, 1978). It is important to note that many of the 76 currently known subtraction bugs were not yet observed back then. One can assume that their model would generate more bugs than the ones reported in (Young & O'Shea, 1981). Section 10.3 discusses their approach in some detail.

A chart comparing the results of the three theories is presented as figure 2-22. Observed bugs that no theory generates are not listed, nor are bugs that have not been observed. (N.B., the figures in Brown & VanLehn (1980) count bugs differently than they way they are counted here. That report counts combinations of bugs with coercions as distinct bugs – see the note on coercions in appendix 2.) The chart shows that the present theory generates more bugs, which is not surprising since it embeds many of the earlier theories' ideas. What is perhaps a little surprising is that there are a few bugs that they generate and it does not. These bugs deserve a closer look.

Y&O	B&V	Cur.	Occurs	Bug
		√	6	Always-Borrow-Left
		√	1	Blank-Instead-of-Borrow
		√	7	Borrow-Across-Second-Zero
√	√	√	41	Borrow-Across-Zero
		√	4	Borrow-Don't-Decrement-Unless-Bottom-Smaller
		√	2	Borrow-From-One-Is-Nine
		√	1	Borrow-From-One-Is-Ten
√	√	√	14	Borrow-From-Zero
√			1	Borrow-From-All-Zero
	√	√	2	Borrow-From-Zero-Is-Ten
	√	√	18	Borrow-No-Decrement
		√	6	Borrow-No-Decrement-Except-Last
		√	1	Borrow-Treat-One-As-Zero
	√		1	Can't-Subtract
		√	1	Doesn't-Borrow-Except-Last
√			15	Diff-0 − N=0
√			43	Diff-0 − N=N
√			1	Diff-N − N=N
√			6	Diff-N − 0=0
		√	7	Don't-Decrement-Zero
		√	4	Forget-Borrow-Over-Blanks
		√	2	N − N-Causes-Borrow
		√	1	Only-Do-Units
	√	√	5	Quit-When-Bottom-Blank
	√		4	Stutter-Subtract
√	√	√	115	Smaller-From-Larger
		√	3	Smaller-From-Larger-Except-Last
	√	√	5	Smaller-From-Larger-Instead-of-Borrow-From-Zero
		√	7	Smaller-From-Larger-Insteadof-Borrow-Unless-Bottom-Smaller
		√	3	Stops-Borrow-At-Multiple-Zero
√	√	√	64	Stops-Borrow-At-Zero
	√	√	1	Top-Instead-of-Borrow-From-ZEro
√	√		5	Zero-Insteadof-Borrow
10	12	25		totals

Figure 2–22
Comparison of observed bugs generated by three theories:
Y&O = Young and O'Shea; B&V = Brown & VanLehn; Cur. = current theory

Early repair theory generates a bug called Stutter-Subtract that the present theory does not generate:

```
Stutter-Subtract:      3 4 5        3 4 5        8 9 7
                     -     2      - 2 2 2      -   6 7
                     -------      -------      -------
                     1 2 3 ×      1 2 3 √      2 3 0 ×
```

This bug does not know how to handle one-digit columns. It impasses when it tries to do such a column. Early repair theory used a repair called Refocus Right to fix the impasse. It would cause the column difference operations to use the nearest digit in the bottom row instead of the blank. Thus, the second column in the first problem is answered with 4 − 2.

The present theory has a non-directional Refocus repair. It finds the fetch pattern responsible for the current focus of attention and rematches the pattern. It finds the closest match that will get the procedure past the impasse. In this case, there is no such match due to the way the grammar structures the subtraction problem. To obtain the equivalent of Refocus Right would require a grammar that views the problem as three multidigit rows instead of a list of three-place columns. Such a grammar would probably generate Stutter-Subtract, but might generate some star bugs as well.

The point behind the Stutter-Subtract story is that early repair theory had some notational knowledge embedded in its repairs. It had several Refocs repairs, and they were specialized for the grid-like notation of subtraction. In the present theory, all knowledge about notation is embedded in the grammar. The Refocus repair is general. It doesn't know about any particular notation. In the early theory, it was stated that the repairs were *specializations* of weak, general-purpose methods. They were tailored for subtraction. In the present theory, the repairs actually *are* general-purpose methods, not specializations.

Young and O'Shea's model generates a class of bugs that they call "pattern errors." At that time, four bugs were included in this class:

Diff-0−N=N	If the top of a column is 0, write the bottom as the answer.
Diff-0−N=0	If the top of a column is 0, write zero as the answer.
Diff-N−0=0	If the bottom of a column is 0, write the zero as the answer.
Diff-N−N=N	If the top and bottom are equal write one of them as the answer.

Young and O'Shea derive all four bugs the same way. Each bug is represented by a production rule, and the rule is simply added to the production system that models the student's behavior. Put differently, they derive the bugs formally by stipulating them, then explain the stipulation informally. Their explanations are:

> The zero-pattern errors are also easily accounted for, since particular pattern-sensitive rules fit naturally into the framework of the existing production system. For example, from his earlier work on addition, the child may well have learned two rules sensitive to zero, NZN and ZNN [two rules that mean N±0=N and 0±N=N]. Included in a production system for subtraction, the first, NZN, will do no harm, but rule ZNN will give rise to errors of the "0−N=N" type. Similar rules would account for the other zero-pattern errors. If the child remembers from addition just that zero is a special case, and that if a zero is present then one copies down as the answer one of the numbers given, then he may well have rules such as NZZ or ZNZ [the rules for the bugs Diff-N−0=0 and Diff-0−N=0].... Rule NNN [the rule for the bug Diff-N−N=N] covers the cases where a child asked for the difference between a digit and itself writes down that same digit. It is clearly another instance of a "pattern" rule. (Young & O'Shea, 1981, pg. 163).

The informal explanations, especially the one for Diff-0−N=N, are plausible. To treat them fully, one would have to explain why *only* the zero rules are transferred from additions, and not the other addition rules.

The point is that one can have as much empirical adequacy as one wishes if the theory is not required to explain its stipulations in a rigorous, formal manner. The present theory could generate the same pattern bugs as Young and O'Shea's model simply by adding the appropriate rules to the AOGs and reiterating their informal derivation (or tell any other story that seems right intuitively). This would not be an explanation of the bugs, but only a restatement of the data embroidered by interesting speculation. This approach does not yield a theory with explanatory value. In short, there is a tradeoff between empirical adequacy and explanatory adequacy. If the model is too easily tailored, then it is the theorist and not the theory that is doing the explaining. The theory per se has little explanatory value. So tailorability, and explanatory adequacy in general, are key issues in evaluating the adequacy of the theory.

Hillclimbing

It would be foolish to claim that the present theory is wonderful because half of its predictions about the Southbay experiment were confirmed. It would be equally foolish to assert that the theory is in desperate need of improvement because it models only two-thirds of the bugs in the Southbay sample. The observational adequacy figures are meaningless in isolation. As an *absolute* measure of theoretical quality, observational adequacy is nearly useless. However, it is excellent as a *relative* measure of theoretical quality. One takes two theories and compares their observational adequacy over the same data, taking care to study their tailorability as well.

Observational adequacy is particularly useful in comparing a new version of the theory to an older version. This allows one to determine whether the new version improves the empirical quality or hurts it. Indeed, this is how the present theory arrived at its current form. To put it in the language of heuristic search (which some claim is a good metaphor for scientific discovery), observation adequacy has been used to *hillclimb:* to find a maximum in the space of possible theories. The claim, therefore, is not that the theory's current degree of observational adequacy is good or bad in an absolute sense, but rather that it is the best that any theory can do, given the same data and the same objectives.

There is a well known problem with hillclimbing. One can get trapped at a local maximum that is not a global maximum. A common solution to this problem is to begin with a gross representation of the landscape so that the search can find the general lay of the land and thereby determine approximately where the global maxima will be. This done, hillclimbing can be done at the original level of detail, but remaining in the limited area where any local maxima are likely to be global maxima as well. The same strategy has been used in this research (or at least, one can reconstruct the actual research history this way). There are three levels of hypotheses (which are also the three levels of organization of the following chapters). The most general level, the architecture level, is a gross representation of the cognitive landscape. It addresses general issues, such as whether learning is basically inductive or not. Hillclimbing in the architectural level yields several hypotheses that define the theory in a non-detailed way. The next lower level of detail is the representation level. It searches through a thicket of knowledge representations issues, e.g., whether procedures should be hierarchical or not. The third level, the bias level, is the last stage of hillclimbing. It finds hypotheses about inductive biases that will optimize the fit between the model's predictions and the data. Because the arguments for hypotheses are structured into gross, medium and fine levels of detail, one can be somewhat assured that the hillclimbing implicit in this strategy has brought the theory to a global maximum.

It bears reiterating that empirical quality is not the only measure of theoretical validity. It must be balanced against explanatory adequacy — does the theory really explain the phenomena or does it just recapitulate them, perhaps because they have been tailored into the model's parameter settings? This theory is quite strong in the explanatory department. The model takes only three inputs, and these inputs are such that the theorist has little ability to tailor the predictions to the data. This implies that the predictions are determined by the structure of the model, which is in turn determined by the hypotheses of the theory. So, the competitive argumentation that fills the remaining chapters can be construed as a hillclimbing adventure where the measure of progress is a combination of increasing observational adequacy and decreasing tailorability.

Chapter 3
Getting Started

The methodological goal of this research is to provide competitive arguments for supporting each hypothesis. But when every hypothesis needs a motivation, and a motivation needs other hypotheses, then getting started is difficult. Some hypotheses must be given support that does not depend on any other hypotheses. These initial hypotheses are often called, somewhat unfairly, assumptions. This chapter states the theory's assumptions, which are two: students learn inductively, and what they learn are procedures. The chapter trys to make these two hypotheses appear plausible in various ways. Later chapters will be able to use these initial hypotheses as the foundations for competitive argumentation; here, there is no such foundation, so hands must be waved.

3.1 Teleology or program?

The first assumption is that student's knowledge about procedures is schematic but not teleological or prototypical. To define these terms, "schematic," "teleological," and "prototypical," several other terms must be introduced. (Figure 3-1 is a road map for the terms that will be introduced.) Computer programmers generally describe a procedure in three ways:

- *Program:* A program is a schematic description of actions. It must be instantiated, by giving it inputs, before it becomes a complete description of a chronological sequence of actions.
- *Action sequences:* One can describe a particular instance of a program as a chronological sequence of actions (or as a sequence of problem states. For the present discussion we'll use action sequences, reserving problem state sequences to serve the same purpose later). That is, executing a program produces an action sequence. In principle, one could describe a procedure (N.B., the term "procedure" is being used temporarily to mean some very abstract, neutral idea about systematic actions) as a possibly infinite set of action sequences. This is analogous to specifying a mathematical function as a set of tuples (e.g., n! as {<0,1>, <1,1>, <2,2>, <3,6>, <4,24>, ...}). Action sequences are not usually used this way. Their most common use in programming practice is in reporting times where a program did something unexpected (i.e., bug reports).
- *Specifications:* Specifications say what a program ought to do. Often they are informally presented in documents that circulate among the programmers and market researchers on a product development team. Sometimes specifications are written in a formal lanuguage so that one can prove that a certain program meets them.

There are names for the processes of transforming information about the procedure from one level to another. *Programming* is the transformation of a specification into a program. *Execution, interpretation* and *running* are names for the transformation of a program into an action sequence. There are also names for static, structural representations of these transformations. A *trace* is a structural representation of the relationship between a program and a particular execution of it. A *procedural net* (Sacerdoti, 1977), a *derivation* (Carbonell, 1983b) and a *planning net* (VanLehn & Brown, 1980) are all formal representations of the relationship between a specification and a program. Actually, these three terms are just a few of the formalisms being used in a rapidly evolving area of investigation. Rich (1981) has concentrated almost exclusively on developing

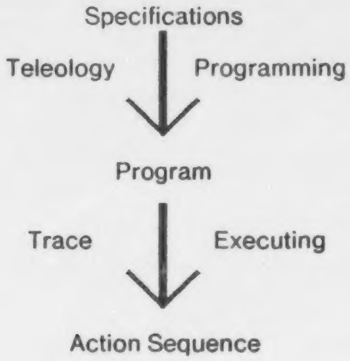


Figure 3–1
Three levels of description for a "procedure." Names for the processes of converting from higher levels to lower levels are one the right. Names for conversion structures are on the left.

a formalism describing the relationship between a specification and a program. In his representation systems, both the specification and the program are *plans* — the surface plan (program) is just a structural refinement of the other. Rather than seeming to commit to one or another of these various formalisms, the neutral term "teleology" will be used. Thus, the teleology of a certain program is information relating the program and its parts to their intended purposes (i.e., to the specification).

Since "teleology" is a new term, it is worth a moment to sketch its meaning. The teleology of a procedure relates the surface structure (program) of the procedure to its goals and other design considerations. The teleology might include, for instance, a goal-subgoal hierarchy. It might indicate which goals serve multiple purposes, and what those purposes are. It might indicate which goals are crucially ordered, and which goals can be executed in parallel. If the program has iterations or recursions, it indicates the relationship between the goals of the iteration body (or recursion step) and the goal of the iteration (recursion) as a whole. In general, the procedure's teleology explicates the *design* behind the procedure.

It is an empirical question which level of description — action sequences, traces, program, teleology or specification — most closely corresponds to the knowledge that student's acquire. It is possible that students could simply memorize the examples that they have seen. In this case, their knowledge of the procedure would be appropriately represented by a set of action sequences. One might call them "prototypes" for the procedure (c.f. theories of natural kind terms based on prototypes, e.g., Roach & Mervis, 1975). However, it is a fact that student's knowledge of mathematical procedures is productive, in the sense that they can solve problems that they have never seen solved. As discussed in section 1.2, accounting for this productivity is problematic when knowledge is represented as sets of action sequences. It will be assumed that student's knowledge is not prototypical. In fact, the only two levels of description that seem at all plausible are programs and teleologies. Finding empirical differences between them is subtle, but not impossible.

Consider, for instance, a procedure for making gravy. A novice cook often knows only the surface structure (program) of the gravy recipe — which ingredients to add in which order. The expert cook will realize that the order is crucial in some cases, but arbitrary in others. The expert also knows the purposes of various parts of the recipe. For instance, the expert understands a certain sequence of steps as making a flour-based thickener. Knowing the goal, the expert can substitute a cornstarch-based thickener for the flour-based one. More generally, knowing the teleology of a procedure allows its user to adapt the procedure to special circumstances (e.g., running out of flour). It also allows the user to debug the procedure. For instance, if the gravy comes out lumpy, the expert cook can infer that something went wrong with the thickener. Knowing which steps of the recipe make the thickener, the cook can discover that the bug is that the flour-fat mixture (the roux) wasn't cooked long enough. The purpose of cooking the roux is to emulsify the flour. Since the sauce was lumpy, this purpose wasn't achieved. By knowing the purposes of the parts of the procedure, people are able to debug, extend, optimize, and adapt their procedures. These added capabilities, beyond merely following (executing) a procedure, can be used to test for a teleological understanding.

Do students acquire the teleology of mathematical procedures?

Gelman and her colleagues (Gelman & Gallistell, 1978; Greeno, Riley & Gelman, forthcoming) used tests based on debugging and extending procedures in order to determine whether children possess the teleology for counting (young children don't, older children do). Adapting their techniques, I tested five adults for possession of teleology for addition and subtraction. All subjects were competent at arithmetic. None were computer programmers. The subjects were given nine tasks. Each task added some extra constraint to the ordinary procedure, thereby forcing the subject to redesign part of the procedure in order to bring it back into conformance with its goals. A simple task, for example, was adding left to right. A more complex task was inventing the equal additions method of borrowing (i.e., the borrow of $53-26$ is performed by adding one to the 2 rather than decrementing the 5). The results were equivocal. One subject was unable to do any of the tasks. The rest were able to do some but not all of the tasks. The experiment served only to eliminate the extremes: Adults don't seem to possess a complete, easily used teleology, but neither are they totally incapable of constructing it (or perhaps recalling it). Further experiments of this kind may provide more definitive results. In particular, it would be interesting to find out if adults were constructing the teleology of the procedure, or whether they already knew it. At any rate, it's clear that not all adults possess operative teleology for their arithmetic procedures, and moreover, some adults seem to possess only surface structures (programs) for accomplishing a task.

Adults found the teleology test so difficult that I was unwilling to subject young children to it. However, there is some indirect evidence that students acquire very little teleology. It concerns the way students react to impasses (i.e., getting stuck while executing a procedure). Consider the decrement-zero impasse discussed in section 2.9. A hypothetical student hasn't yet learned how to borrow from zero although borrowing from non-zero numbers is quite familiar. Given the problem

$$\begin{array}{r} 6\ 0\ 4 \\ -\ 2\ 1\ 7 \\ \hline \end{array}$$

the student starts to borrow, attempts to decrement the zero, and reaches an impasse. If the student understands the teleology of borrowing, then the student understands that borrowing from the hundreds would be an appropriate way to fix the impasse. The purpose of borrowing is to increase a certain place's value while maintaining the value of the whole number. Here, the tens place needs to be increased so that it can be decremented. Borrowing will serve this purpose. In short, the teleology of non-zero borrowing allows it to be easily extended to cover borrowing from zero. Although some students may react to the decrement-zero impasse this way, many do not. They use local problem solving instead, as discussed in section 2.9. Because students do not make teleologically appropriate responses to impasses, it appears that they did not acquire much teleology (or if they do, they are unwilling to use it — in which case it's a moot point whether they have it or not).

Mathematical procedures are perhaps a little different than other human procedure in that their teleology is quite complex. The complexity is due partly to the fact that the procedures manipulate a representation (e.g., base-10 numerals) rather than the objects of interest themselves (e.g., numbers). A procedure for making gravy does not have this problem. Cooks don't manipulate representations of flour and water, they manipulate the real stuff. An added complexity in arithmetic procedure is way their teleologies merge loops to accomplish several goals at once (VanLehn & Brown, 1980). The teleology of loops is so complex that only recently has AI made much progress on analyzing it (Waters, 1978; Rich, 1981). In other task domains than learning mathematical procedures, more students might show evidence of teleological knowledge. In the present domain, it is safe to assume that students knowledge is more like a program than a teleology. That is, the knowledge is schematic rather than teleologic (or prototypical).

3.2 What kind of learning goes on in the classroom?

The second assumption made by the theory is that students acquire their procedures by induction: they generalize from examples. This section trys to make that assumption seem plausible. First, it shows that inductive learning is consistent with the gross features of the students' classroom experiences. Then, it presents several other ways that procedures could be learned, and casts a little doubt on each of them.

No one knows precisely what goes on in elementary school. Unlike college classes, elementary school classes are not just lectures and recitations. For much of the day, the child *lives* in the school classroom. Many activities that go on there have little to do with learning. Schools are like business offices in this respect. Despite the fact that both schools and offices have ostensive purposes, it is impossible to precisely describe all the activities going on inside their walls. However, the gross features of classroom setting are uncontroversial.

In elementary school, math is taught once a day for a little less than an hour, usually in the morning when children are least restless. The most common instructional activity is *seatwork*: the students work exercises in their seats, occasionally asking the teacher for help. Figure 3–2 shows how one study of math classes divides instruction time. It excludes non-instructional activities such as collecting homework, dividing into groups, and dealing with disciplinary problems. The largest proportion of instructional time is spent in seatwork.

Activity	Grade 2	Grade 5
seatwork	55%	70%
discussion, recital	30%	20%
lecture, demonstration	10%	5%
games	5%	5%
total	100%	100%

Figure 3–2
Gross proportions of time spent in various instructional activities.
Adapted from Ramos-4 data reported in (McDonald & Elias, 1975).

Most teachers follow the lesson plans of the textbook rather closely. The contents of the teacher's textbook can be taken as a rough approximation to the material that the teacher actually presents. Judging from the textbooks, the calculational procedures that I am calling "mathematical skills" are only a fraction of the mathematical curriculum. Moreover, they are not taught in a nice compact unit as one might expect from college curricula. The lessons that introduce the components of a procedure are scattered over several years. In the Scott-Foresman textbook, for instance, the first lessons on the multicolumn subtraction procedure occur midway through second grade. The various subprocedures, such as traversing columns and borrowing, are introduced in six chapters scattered through the last half of the second grade, the third grade, and the first half of the fourth grade. A chapter typically has one or two lessons that introduces a new subskill, several lessons reviewing previously taught subskills, and a chapter test. During the two years that the subtraction procedure is actively taught (it is reviewed for many years thereafter), the students cover about 600 pages of text. At most 90 pages directly address the subtraction procedure. These 90 pages include not only the lessons introducing new subskills, but also review lessons and tests. Page counts can be translated into a rough measure of the time spent learning subtraction. If one puts the school year at about 175 days of usable instruction, and math occupies an hour a day, then it works out that the subtraction procedure is taught in about 50 hours. These are just rough estimates, of course. The main points are that the subtraction procedure does not consume much instructional time, that most of that time is spent on review, and that the skill in introduced gradually over a long period. The same comments apply to other calculation skills. Algebra equation solving is introduced in the fifth grade, in the Scott-Foresman textbook series mentioned above. By the time the student takes high school algebra, most of linear equation solving has already been presented.*

An inductive account of skill acquisition requires that the curriculum provides examples in appropriate quantities and varieties. Textbooks and teachers provide some worked examples of mathematical procedures, but not all that many. A typical borrow lesson in a textbook might print two worked examples and 25 exercises. The teacher will undoubtedly work through several of the exercises on the blackboard with the class and leave some of them on the chalkboard while the students do their seatwork. So a lesson might have a half dozen or a dozen examples for the students to generalize from. The example set is not as small as one or two, but it is not hundreds either. One question that a computational theory can address, in detail, is whether this moderate number of examples has sufficient information for induction to succeed. To the first order, however, it seems that the instruction in use today has enough examples with enough variety to make an inductive account of learning plausible.

---

* It is important to consider the whole curriculum when grounding a learning model on textbook lessons. In particular, it is easy to mistake a review lesson for an introductory lesson. Neves (1981) built an AI program, ALEX, that learned how to solve simple algebra equations by induction. Neves tested ALEX using examples abstracted from a high school algebra textbook. The first algebra lesson that Neves used to test ALEX is probably a review lesson. It presents three operators for solving linear equations, all on one page. One of these operators is taught as early as the fifth grade in the Scott-Foresman series. Neves has ALEX learn these three operators from scratch, as if this lesson were introducing them for the first time. I believe this is a mistake that caused Neves to make ALEX too powerful to be plausible as a model for the initial acquisition of procedures (see sect. 4.3).

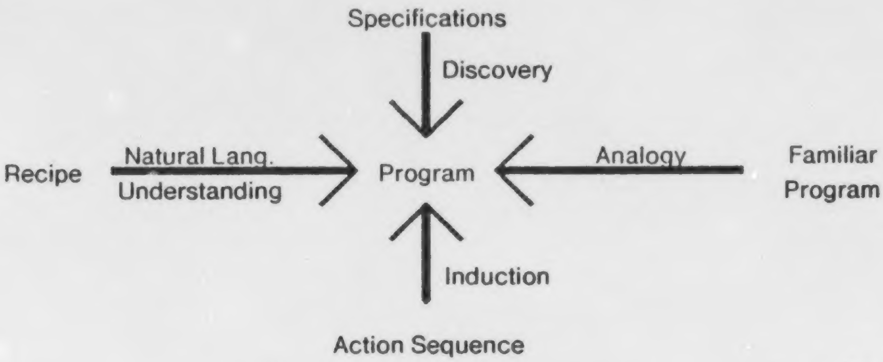


Figure 3–3
Ways to acquire a program level of description.

The gross features of classroom life seem consistent with an inductive account of learning. But they are also consistent with just about any other account of learning because classrooms are a rich, almost chaotic environment. Since the assumption of inductive learning will soon be used in important ways, it is worth casting some doubt on the competing accounts. In order to organize them somewhat, we'll again use the tripartite distinction between specifications, programs and actions sequences. If the goal is to construct a description of the procedure at a program (schematic) level, there are four possible routes (see figure 3–3):

1. From specification to program: A kind of learning by doing or discovery.
2. From examples (action sequences) to programs: induction.
3. From some other schematic description, either
   (a) another familiar program: learning by analogy, or
   (b) a natural language presentation of the program.

Each of the four routes has a certain degree of plausibility with respect to the gross features of classroom life. Discovery learning would take place while the students solve problems alone, cogitating over their mistakes. There is ample opportunity to do this in a class where 55% of the time is spent doing seatwork. Inductive learning requires examples, which the teacher and the text supply. Analogic learning requires the juxtaposition of familiar procedures with the target procedures. Modern instruction does some of this by drawing analogies to monetary transactions or games involving other concrete numerals (e.g., Dienes blocks, Montessori rods, abacci, etc.). Learning by understanding natural language presentations of procedures has some support in that recipe-like presentations of procedures are occasionally used in textbooks and (presumably) classrooms. So all four routes have a certain degree of plausibility.

The next three sections will take three of these four accounts of learning in turn, leaving induction aside, and show why each is not a plausible account of the way mathematical procedures are acquired. In the process, several of the pertinent AI studies of procedure acquisition will be mentioned (for a comprehensive review, see Cohen & Feigenbaum, 1983). By the way, the following remarks should not be construed as claims that inductive instruction is the only effective kind. Indeed, it may be that other kinds of instruction are so effective that the few students who actually utilize the instruction don't have bugs, and hence there would be no sign of their learning the bug data (I doubt this very much). It may be that inductive instruction is not nearly as good as some other method in that decreasing the curricular emphasis on examples would improve students' learning. The point is that this is not a theory about what learning *should* occur, it is only a theory about what learning *does* occur.

3.3 Learning by discovery

In discovery learning, students learn on their own by solving problems. The teacher has little direct involvement, except perhaps to suggest projects or problems for the students to tackle, or to answer an occasional question. The key assumption of discovery learning is that the students can solve problems initially. They may only be able to solve simple problems. They may solve them by trial and error, making many counterproductive moves in the processes. Discovery learning requires at least this much initial competence of its participants. By solving problems on their own, students discover ways to avoid counterproductive moves and ways to solve problems that they couldn't solve initially.

Superficially, it appears that discovery learning is common in current mathematical education. A typical lesson has exercises that are just a little bit harder than the ones in the preceding lesson. Such a gradual increase in the level of difficulty seems just right for encouraging a discovery learner to acquire the new subskill that the lesson teaches. One can imagine, for example, a lesson that teaches carrying with exercises such as *a*:

```
a.          3 5              b.          3 5
          + 2 8                        + 2 1
          -----                        -----
```

Exercise *a* is just a little harder than *b*, a problem type that the hypothetical discovery learner has presumably mastered already. The learner would attack *a*, first generating *c*. She would recognize that the answer of *c* isn't a proper number, so she would fix it, yielding *d*.

```
c.          3 5              d.          3 5
          + 2 8                        + 2 8
          -----                        -----
            5 13                         5 13
                                         6 3
```

This results in a procedure with two passes: one pass adds the columns, the second pass converts the answer to the canonical form for numbers. Analyzing this solution and others like it may eventually lead the learner to discover that the two passes can be merged into one. This merger would generate the normal add-with-carry procedure. So, it seems that discovery learning is quite compatible with the kinds of instruction that today's students receive. However, when one looks a little closer, this illusion disappears.

The key requirement of discovery learning is that students know enough about the task that they can solve it initially, albeit in an inefficient way. This requirement is not a peculiarity of people, but an apparently necessary part of the information processing. All AI discovery learners* have been equipped with substantial initial domain knowledge. For instance, LEX is a program that discovers how to efficiently solve simple integrals (Mitchell et. al., 1983). However, its initial knowledge of the domain contains a complete set of legal mathematical transformations (e.g., integration by parts). This much knowledge is quite a bit more than any beginning calculus student has. Another well-known discovery learner is Sussman's HACKER program (Sussman, 1976). It learns procedures for stacking toy blocks. However, its initial state of knowledge contains a "physics" for the blocks world (e.g., two blocks cannot occupy the same place). This knowledge is essential for detecting when the partially learned procedure has bugs. Without it HACKER would not know to revise its procedures. It would not learn. Similar comments apply to almost all other AI discovery learners. To put it in terms used before, discovery learners must have a specification of the procedure (in some form). From that they can derive a program for the procedure.

---

* Except Lenat's AM program, which uses a representation in which relevant task domain information was unusally dense (Lenat & Brown, 1983). By "AI discovery learners," I mean programs for planning (e.g., Sacerdoti, 1977; Green & Barstow, 1975; Stefik, 1980), strategy acquisition (e.g., Hayes-Roth & McDermott, 1976), operalization (e.g., Mostow, 1981), learning by doing (e.g., Anderson, 1980; 1982; Anzai, 1979) and learning by debugging (e.g., Sussman, 1976).

As noted earlier, students of mathematical procedures seem not to possess the teleology of their procedures. However, without at least the specifications for a procedure, they cannot perform discovery learning. For instance, most young students do not understand the base-10 number system well enough to see that the answer in problem *c* above is not a legal number. They lack the kind of knowledge that HACKER used to detect when its procedure had a bug. Without this kind of knowledge, there is no way students can discover the carrying subskill on their own.

In the arithmetic domain, the essential problem is that the specifications for procedures must be couched in terms of preserving relationships between *numbers*, but the procedures manipulate base-10 *numerals*. More generally, all mathematical calculations manipulate symbols and not what the symbols stand for. Since the symbols do not necessarily obey constraints that preserve their semantics, the student must *know* not to violate these constraints. The symbols will not themselves prevent a student from creating buggy procedures in the way that HACKER's blocks prevent it from creating buggy procedures.

Many educators and cognitive scientists have tried to find ways to teach mathematical notation that will enable mathematical calculations to be learned by discovery. A typical technique involves substituting physical objects for the symbols of the notation. Constraints on the notation are turned into physical constraints on the objects. For instance, I once tried to use this technique to get young children to discover carrying. The basic idea was to make the principles of the base-10 system extremely salient by using an appropriate physical representation for numbers. A two digit number was represented by two egg cartons that were trimmed to hold just nine eggs. If the student tried to put more than nine eggs in the units carton, they would roll off the table and break. The idea was to convert a tacit constraint of the base-10 system, that the maximum place-value holder was nine, into an extremely salient physical constraint. Betsey Summers and I tried to coax eight beginning arithmetic students to synthesize carrying. Not one would do it. After they were shown the procedure, they would perform it with no trouble (i.e., they learned it by induction). Since the constraints defining the task were salient, their failure can only be attributed to an inability (or perhaps unwillingness) to do the kind of problem solving that discovery learning requires. I hasten to add that this experiment should not be taken as definitive. Young subjects present difficult methodological problems. By changing the instructions or the experimental materials, one can vastly alter the apparent competence of the subjects (c.f. Klahr and Robinson's study of the Tower of Hanoi, 1981, or Gelman and Galistrel's "magic" experiments, 1978). Resnick and others have reported more methodical experiments of this kind where some students were able to discover carrying, borrowing and similar subprocedures (Resnick, 1982). Nonetheless, the general consensus is that it is difficult and time consuming to teach enough about the semantics of the notation that students can learn calculations by discovery. It seems safe to assume that little or no discovery learning occurs in the typical classroom.

3.4 Learning by analogy

Learning by analogy is the mapping of knowledge from one domain over to the target domain, where it is applied to solve problems. Winston (1979) showed that learning procedures by analogy could be formalized. He constructed a program that could learn to solve Ohm's law problems by drawing an analogy to hydraulics (see also Brown, 1977; Carbonell, 1983a; 1983b). Analogies are heavily used in the early grades to teach base-10 numeration. Students are often drilled on the mapping between written numerals and and various concrete representations of numbers, such as collections of coins, Diennes blocks, Montessori rods and so forth. This is a mapping between two kinds of numerals, and not two procedures. Later, this inter-numeral mapping is appealed to in teaching carrying and borrowing. For example, a known procedure for making change — trading a dime for ten pennies — is mapped into the borrowing procedure of written subtraction. Since this kind of teaching is quite common in the primary grades, it seems quite plausible that learning by analogy should be a prominent framework for learning procedures.

Presumably, once an analogy has transferred some knowledge, it is still available for use later to transfer more knowledge about the procedure. In some cases, this predicts significant student competence. For instance, if the students learned simple borrowing via the analogy, then it's quite plausible that when confronted with more complex borrowing problems, such as

```
  6 0 7
- 2 3 8
```

(assuming the student hasn't yet been taught how to solve such borrow across zero problems), the student could solve the problem in the concrete domain by trading a dollar for nine dimes and ten pennies, then map back into the written domain, thus producing the correct solution. Indeed, the analogies used in instruction may have been designed so that these productive extensions of the base analogy are encouraged.

But this is a much more productive understanding of borrowing than most students achieve. As discussed in the preceding chapter, when certain students discover that it is impossible to decrement the zero, they will do local problem solving — repairing their execution state. These students do not use analogies to familiar procedures (e.g., making change). If the students had learned their procedures via analogy, one would have to make ad hoc stipulations to explain why they no longer used that analogy after they had learned the procedure. It's more plausible that they simply didn't utilize the analogy in the first place. Similar comments apply to the analogies between arithmetic and algebra. They would predict more algebraic competence than one typically finds. Loosely speaking, learning by analogy is too good. It predicts that students would "repair" impasses by constructing a correct extension to their current procedure. That is, they would *debug* instead of repair. Since some students do apparently have repair-generated bugs, another explanation would be needed for how these students acquired their procedures. At the very least, analogy cannot be the only kind of learning going on, if it happens much at all.

Carbonell (1983) makes a telling argument about analogies between procedures. His ARIES program was unable to form analogies between certain procedures when all it had was the program (schematic) representations. However, Carbonell found that analogies could be forged when the procedures were described teleologically (i.e., in Carbonell's terminology, the analogy is between *derivations* of procedures). Suppose one stretches Carbonnell's results a little and claims that knowing the teleology (derivation) of procedures is *necessary* for procedural analogy, at least for mathematical procedures. (Carbonnell claims only sufficiency, if that.) Since most math students are ignorant of the teleology of their procedures (section 3.1), one can conclude that students did not acquire their procedures via analogy.

Analogies are hard to make

How is it that teachers can present material that is specifically designed to encourage learning procedures by analogy, and yet their students show few signs of doing so? Winston's research (Winston, 1979) yields a speculative answer. It indicates that the most computation intensive part of analogy can be discovering how best to match the parts of the two sides of the analogy. To solve electrical problems given hydraulic knowledge, one must match voltage, electrical current and resistance to one each of pressure, water current and pipe size. There are 6 possible matchings, and only one matching is correct. The number of possible matchings rises exponentially with the number of parts. For a similar analogy, a best match had to be selected from 11! or 40 million possible matches. The matching problem of analogy is a version of a well known NP-complete problem: finding the maximal common subgraph of two digraphs (Hayes-Roth & McDermott, 1978). Hence, it is doubtful that a faster solution than an exponential one exists.

If computational complexity can be equated with cognitive difficulty, Winston's work would predict that students would find it difficult to draw an analogy unless either it was a very simple one or they were given some help in finding the matching. Resnick (1982) has produced some experimental evidence supporting this prediction in the domain of mathematical instruction. Resnick interviewed students who were taught addition and subtraction in school, using the usual analogies between concrete and written numerals. It was discovered that some students had mastered both the numeral analogy and the arithmetic procedures in the concrete domain, and yet they could not make a connection between the concrete procedures and the written ones. Resnick went on to demonstrate that students could easily make the mapping between the two procedures provided that the steps of the two procedures were explicitly paired. That is, the student was walked through the concrete procedure in parallel with the written one. A step in one was immediately followed by the corresponding step(s) in the other. If we assume the conjecture from above, that combinatorial explosions in mapping equates with difficulty for humans making analogies, and we assume that "parts" of procedures roughly correspond to steps, then Resnick's finding makes perfect sense. The procedures are currently presented in school in a non-parallel mode. This forces students to solve the matching problem, and most seem unable to do so. Consequently, the analogy does little good. Only when the instruction helps the students make the matching, as it did in Resnick experiment, does the analogy actually succeed in transferring knowledge about one procedure to the other. In short, analogy could become a major learning technique, but current instructional practices must be changed to do so.

Example-exercise analogies

There is anecdotal evidence that analogy is very common, but it is analogy of a very different kind. In tutoring, I have watched students flip through the textbook to locate a worked problem that is similar to the one they are currently trying to solve. They then draw a mapping of some kind between the worked problem and their problem that enables them to solve their problem. Anderson et. al. report the same behavior for students solving geometry problems (1981). Although the usage could be disputed, Anderson et. al. call this kind of example-exercise mapping an analogy. It differs from the kind of analogy discussed earlier. The abstraction that is common to the two problem solutions is exactly the surface structure (program) of the procedure. In the analogy between making change and borrowing, the common abstraction lay much deeper, somewhere in the teleology of the procedure. To put it differently, the example-exercise analogy maps two action sequences of a procedure together, thus illustrating the procedure's program. The other analogy maps two distinct procedures together in order to illustrate a common teleology.

The former mapping, between two instances of a schematic object, is nearly identical to the central operation of learning by examples. In both cases, the most specific common generalization of the two instances is calculated. Winston also points out the equivalence of generalization and analogy in such circumstances (Winston, 1979). Although I have not investigated example-exercise analogy in detail, I expect it to behave almost indistinguishably from learning by generalizing examples.

To summarize, one form of analogy (if it could be called that) is indistinguishable from induction. The other form of analogy seems necessarily to involve the teleology of procedures. Since students show little evidence of teleology, it is safe to assume that analogic learning is not common in classrooms, perhaps because current instructional practices aren't encouraging it in quite the right way.

3.5 Learning by being told

One framework for acquiring a procedure involves following a set of natural language instructions until the procedure is committed to memory. This framework for explaining learning is called *learning by being told.* It views the central problem of learning as one of natural language understanding. The key assumption is that the text describes the procedure in enough detail that all the students need to do is understand the language, then they will be able to perform the procedure.

Manuals of procedures are ubiquitous in adult life. Examples are cookbooks, user guides, repair shop manuals and office procedure manuals. In using procedure manuals, adults sometimes learn the procedures described therein, and cease to use the manuals. So learning by being told is probably quite common among adults. The content of procedure manuals can be taken as a model for how good a natural language description has to be if it is to be effective in teaching the procedure.

Open any arithmetic text, and one immediately sees that it is not much like a cookbook or an auto repair manual. There is very little text. The books are mostly exercises and worked examples. The reason is obvious: since students in the primary grades are just beginning to read, they could make little use of an elaborate written procedure.

Badre (1972) built an AI program that reads the prose and examples of a fourth grade arithmetic textbook in order to learn procedures for multicolumn addition and subtraction. Badre sought in vain for simple, concise statements of arithmetic procedures that he could use as input to his natural language understanding program. He comments:

> During the preliminary work of problem definition, we looked for a textbook that would explain arithmetic operations as a clearly stated set of rules. The extensive efforts in this search led to the following, somewhat surprising result: nowadays, young American grade-school children are never told how to perform addition or subtraction in a general way. They are supposed to infer the general algorithms from examples. Thus actual texts are usually composed of a series of short illustrated 'stories.' Each story describes an example of the execution of the addition or the subtraction algorithms. (Badre, 1972, pp. 1-2)

Despite the fact that Badre's program "reads" the textbook's "stories" in order to obtain a description of the examples, the role of reading in its learning is minimal. The heart of the program is generalization of examples. In particular, the program employs only a few heuristics that use the book's prose to disambiguate choices left open by generalization.

The preceding paragraphs discussed primary school students learning arithmetic. Algebra learners are secondary school students. Many can read well enough to use procedure manuals. In secondary school algebra texts, one sometimes finds "recipes" for solving equations and the like, but they are often too terse and ambiguous to serve as more than a simple reminder. Their level of detail suggests that such written procedures are used as *summaries* and not as the primary exposition. The fact that most of them are placed at the end of their chapters suggests that the textbook writers also see them as playing a secondary, summarizing role. If I may add anecdotal support from experiences as an algebra tutor, I have observed that students who flip through the textbook looking for help in solving a problem virtually always refer to a worked example rather than a recipe. This is consistent with the view that recipes play an integrative or summarizing role. They lack the detail to serve either as the main exposition of the procedure or even as a reference when additional details are sought.

3.6 Summary and formalization

This chapter presented two hypotheses. The theory needs them in order to get started in arguing competitively. Since there are no independently motivated hypotheses that can be used in arguing for these two, they can only be justified by making them seem plausible. They are, in this special sense, assumptions — hypotheses without proper support, but ones that the theory bears allegiance to.

One hypothesis is that the knowledge that students acquire is schematic (at the level of a program) rather than teleologic (at the level of a specification for a program) or prototypical (at the level of a set of problem state sequences). All three descriptive levels are logically sufficient to describe a procedure. However, the behavior of students seems best to fit the hypothesis that their descriptions of procedures are schematic. The argument against prototypical knowledge is that students' problem-solving ability is much more productive, in the sense that they can solve problems that they have never seen before, than an account of procedural knowledge that is based on prototypes (i.e., memorized examples) would predict. On the other hand, if students possessed the teleology of their procedures, most impasses could be "repaired" by deriving a correct procedure (i.e., students would debug instead of repair). At least some students, the ones with bugs, must be lacking such teleological knowledge. Also, there is experimental evidence that some adults have no teleology for their arithmetic procedures. They either never learned it or they forgot it in such a way that the schematic level (program) was retained while the teleology was forgotten. All in all, it is more parsimonious to assume that students learn just the schematic level descriptions for their procedures. This implies that student's knowledge can be formalized by something like Lisp procedure or production systems. It is not necessary to use more powerful formalisms such as planning nets (VanLehn & Brown, 1980), planning calculi (Rich, 1981) or procedural nets (Sacerdoti, 1977).

The second assumption is that students learn inductively. They generalize examples. There are several less plausible ways that procedures could be learned: (1) Learning-by-being-told explains procedure acquisition as the conversion of an external natural language information source, e.g., from a procedural manual, into an internal comprehension of the procedure. It is implausible in this domain because young students don't read well and older students' textbooks are not procedure manuals. (2) Learning-by-analogy is used in current mathematical curricula, but in ways that would produce an overly teleological understanding of the procedural skills. If students really understood the analogies, they wouldn't develop the bugs that they do. (3) Discovery learning requires that students have enough initial knowledge of the task that they can muddle through to a solution. Discovery learning describes how students develop solution procedures from their initial, trial-and-error problem solving. However, mathematical tasks have difficult specifications that make it unlikely that a student would blunder into a correct solution of, e.g, a subtraction problem. Even when these specifications are made salient, there is some experimental evidence indicating that the initial trial-and-error problem solving is too hard for many students. Besides, if students did use discovery learning, their knowledge of the procedures would be overly teleological. Of the various ways to learn procedures, only induction seems both to fit the facts of classroom life and to account for the schematic (program) level of knowledge that students appear to acquire.

Formalizing the hypotheses using constraints on undefined functions

Two functions, named `Learn` and `Cycle`, will be used to formalize the theory. The functions will not be defined. Instead, they will gradually acquire meaning as the hypotheses of the theory are stated in term of them. In the next chapter, for instance, `Learn` will be defined in terms of three new undefined functions, and some constraints will be added concerning how the three functions interact. In effect, these new functions and constraints will be a partial definition of `Learn`. Later chapters will introduce further constraints. When all constraints have been made, there will still be many ways to define the various functions. Sierra provides one definition for each. The intention is that any other definition would do as well as Sierra does at predicting the data.

To put it a little differently, the endeaver in the following chapters is to accumulate a set of semi-formal specifications for Sierra. As new empirical facts come to light, new specifications must be imposed on Sierra in order that its performance corresponds to the new facts. To put it baldly, the endeaver is to uncover a *teleology* for Sierra. Chapter 2 presented Sierra at the schematic (program) level. The remaining chapters build up its teleology. The structure of interlocking competitive arguments is exactly a teleology for Sierra, except that it stops short of the actual code itself. It is a teleology for a class of Sierra-equivalents.

The following is a list of the nomenclature, with comments on their intending meanings.

$L_1 \ldots L_i \ldots L_n$	A sequence of lessons.
`L`	A variable designating a lesson.
`P`	A variable designating the student's procedure.
`(Examples L)`	A function that returns the examples contained in its argument, which is a lesson.
`(Exercise L)`	A function that returns the exercises contained in its argument, which is a lesson.
`(Learn P L)`	An undefined function that returns a set of procedures. Its first argument, `P`, is a procedure, and its second argument, `L`, is a lesson. It represents the various ways that its input procedure can be augmented to assimilate the lesson.
`S`	A variable designating the current runtime state.
`(Internal S)`	A function that returns the internal (execution, or interpreter) state.
`(External S)`	A function that returns the external (problem) state. The current state is a composite of the internal and external state.
`(Cycle P S)`	An undefined function that inputs a procedure and a runtime state and outputs a set of possible the next states. It represents one cycle in the interpretation/execution of the procedure.

Defining inductive learning is easy if one can use a consistency constraint such as the following one: If L is a lesson, and g is a generalization induced from it, then g must be consistent with all the examples in L. In the case of procedures, a procedure g is consistent with an example if its solution to the example's exercise problem is the same problem state sequence as the example's problem state sequence. That is, consistency means a student procedure solves the lesson's example exercises using the same writing actions that the teacher did. Given a lesson sequence, $L_1 \ldots L_i \ldots L_n$, the set of observable procedures is obtained by chaining. That is, procedure P_i is learnable when it is induced from procedure P_{i-1} during lesson L_i, and P_i is consistent with lesson L_i, and P_{i-1} is learnable. This recursive definition defines the set of learnable procedures to include ones that are intermediate procedures as well as the procedures that the learners have when they reach the end of the lesson sequence. Although it might appear a little complicated, there is nothing special going on. This is an ordinary way to formalize incremental induction. The formal hypotheses are:

> *Incremental Learning*
> Given a lesson sequence $L_1 \ldots L_n$ and an initial procedure P_0:
> Procedure P_i is a core procedure if and only if
> (1) $P_i = P_0$, or
> (2) $P_i \in$ `(Learn` L_i P_{i-1}`)` and P_{i-1} is core procedure.
>
> *Induction*
> If $P_i \in$ `(Learn` L_i P_{i-1}`)` then for each example problem x in `(Examples` L_i`)`, the problem state sequence that is P_i's solution to x is equal to the problem state sequence that is the solution to x used in the example.

These hypotheses express the second assumption presented in this chapter, that learning is inductive in this domain. The first assumption is not as easy to formalize. There is no standard way to formally distinguish between a schematic procedure (program) and a teleologically described procedure (a teleology). The best that can be done is to appeal to the intuitive notion of an runtime state (notated `S`). It changes during problem solving while another information structure, the procedure (notated `P`) does not change. Later on, this inability to express the assumption formally won't matter because a formal representation for procedures will have been defined. The following principle, as well as the notation defined above, is aimed at providing a foundation for the definition of the knowledge representation and its use in problem solving.

> *Predictions*
> If S_0 is the initial state such that $(\texttt{External}\ S_0)$ is a test exercise, then the set of predicted problem state sequence for students with procedure `P` is exactly the set
> $\{\langle(\texttt{External}\ S_0)\ldots(\texttt{External}\ S_n)\rangle \mid \forall i\ S_i \in (\texttt{Cycle}\ P\ S_{i-1})\}$.

This constraint puts the teeth into the whole theory. It connects observable, testable predictions with the predicted procedures. It defines the solution of a problem to be a sequence of runtime states S_i. Since `Cycle` is non-deterministic in that it can output more than one runtime state, many solution sequences are possible for the same procedure `P` and the same test problem $(\texttt{External}\ S_0)$. Note that only the sequence of problem states, the projection of the S_i via `External`, is observable. It is just barely worth mentioning that this formalization ducks the minor issues of initial and final internal states.

Chapter 4

The Disjunction Problem

This chapter argues that a key problem which any inducer faces is controlling disjunction. If the class of all generalizations is specified in such a way that disjunctions are unconstrained, then an inducer will be unable to identify which generalization it is being taught even if it is given an infinite number of positive and negative examples. It is only when the inducer is given *all possible examples* that it can succeed. This is physically impossible in most interesting domains. Any physically realized inducer that can learn successfully must be performing induction under some set of constraints on disjunctions. In the previous chapter, it was assumed that students learn mathematical procedures inductively. Since they don't require infinitely many examples to do so, there must be constraints on the way disjunctions are induced. The task of this chapter is to find out what those constraints are.

Research on machine induction has discovered several methods that solve the disjunction problem and thus enable mechanical inducers to succeed in finite time. For instance, two classic methods are to bar disjunctions from generalizations or to bias the inducer against generalizations that use disjunctions. These methods, or any methods that solve the disjunction problem, *could* be the one used by people. It is an empirical question which method people *actually* use. It will be shown that the method used by students in this domain is the one-disjunct-per-lesson felicity condition, which was mentioned in chapter 1. Before arguing in defense of the felicity condition, the disjunction problem will be introduced and several solutions will be discussed.

4.1 An introduction to the disjunction problem

By "disjunction," I mean the following: Suppose that g and g' are two generalizations from the class of all possible generalizations. They each have an extension, where a generalization's extension is the set of all possible examples (instances) consistent with the generalization. The generalization is a generalization of each object in its extension and no other objects. A generalization's extension is usually an infinite set. Let x and x' be the extensions of g and g', respectively. The disjunction of g and g' is any generalization whose extension is the union of x and x'.

It is often the case that a representation language is used to define the class of all possible generalizations. If so, disjunction usually corresponds to certain operators or constructions in the representation language. Disjunction takes two or more generalizations and produces a new one such that the extension of the new generalization is exactly the union of the extensions of the old generalizations. For instance, the disjunction of two predicates, e.g., `(WEDGE x)` and `(BRICK x)`, is their logical disjunction, `(WEDGE x) V (BRICK x)`. The disjunction of two context-free grammars is a grammar whose rule set is the concatenation of their two rule sets (assuming all non-terminals except the root have distinct names).

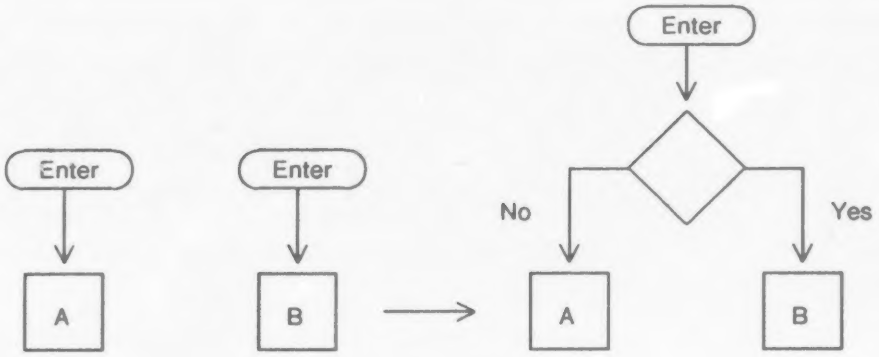


Figure 4–1
Disjunction of two flow charts.

The disjunction of two procedures A and B is their concatenation plus the new top-level statement: "If such-and-such is true then call A else call B." To see this in a little more detail, suppose that procedures are represented as flow charts (see figure 4–1). Each flow chart has a designated node, labeled `Enter`. To disjoin two flow charts, a conditional branch is placed between the two `Enter` nodes of the two flow charts. This forms a single, new flow chart. The test (predicate) inside the conditional branch could be arbitrary (i.e., a random true-false generator) or it could be something specific. It doesn't matter as long as the extension of the new flow chart is the union of the extensions of the two old flow charts. The extension of a flow chart could be considered to be the set of all action sequences (or equivalently, as the set of all problem state sequences. Denotational semantics provides a more general and rigorous treatment of extensions of procedures. See Stoy, 1977).

If the two flow charts were very similar, then the same effect could be achieved by merging them. For instance, suppose the two flow charts were identical except for one conditional branch's test (see figure 4–2). In one flow chart, the test is `P`; in the other flow chart, the test is `Q`. Given this similarity, the disjunction is a flow chart with `(OR P Q)` as its test. Disjunction in the procedural domain has a rather broad interpretation. It can introduce new control structure or just modify internal tests and actions.

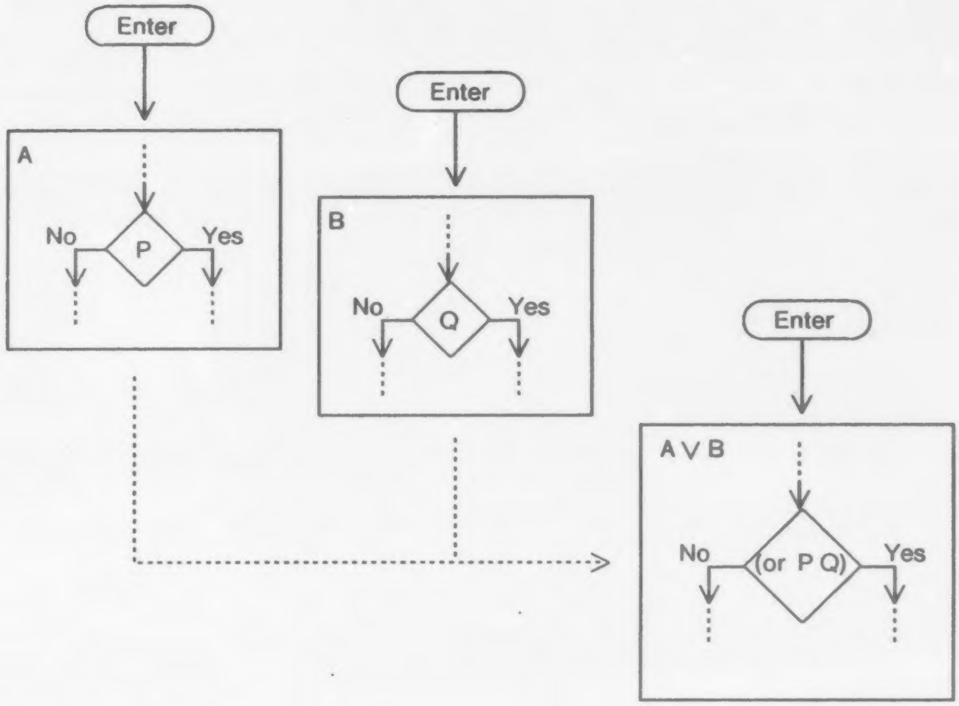


Figure 4–2
Internal disjunction of two flow charts.

The disjunction problem

Induction's trouble occurs when the class of all possible generalizations admits free disjunction. That is, the disjunction of g with g' is in the class whenever g and g' are. When this is the case, induction acquires some strange properties that make it seem unlikely as a form of human learning.

Free use of disjunction allows induction to generate absurdly specific generalizations. One such absurdity is the *trivially specific* generalization: a disjunction whose disjuncts are exactly the positive examples that the learner received. Thus, if the inducer received positive examples *a*, *b* and *c*, then the disjunction `(OR a b c)` is the trivially specific generalization. It has three disjuncts, namely the three examples (or rather, complete descriptions of each example). The trivially specific generalization is not really a generalization at all. Its extension is just $\{a\ b\ c\}$. The inducer didn't really generalize, it just remembered. Clearly, this is not the only kind of knowledge acquisition that people do, especially students of mathematical skills. There must be some constraints on induction that people have. For instance, they could be biased against trivially specific generalizations. This would take care of one problem, but there is another problem that is much worse. It is the heart of the disjunction problem.

When disjunctions are unconstrained, the inducer has to be given the *complete* extension of the generalization being taught before it can reliably discriminate that generalization from the others. To see this, first assume that for each possible example, there is a generalization in the class of all possible generalizations whose extension is that example and only that example. (If this assumption is not true, then we can reformulate the example space into a space of equivalence classes such that the examples in a class cannot be distinguished by generalizations.) For instance, a grammar consisting only of the rule S→*a* is such a generalization, where S is the root category and *a* is a string of terminals. This grammar's extension is the singleton set {a}. Using such singleton generalizations and disjunction, *any finite set of examples can be described by some generalization.* To get the generalization for {a, b}, one finds the generalizations for {a} and for {b}, then takes their disjunction. Since all sets of examples correspond to generalizations, the inducer can't tell which generalization is correct until it is told exactly what the target generalization's extension is. This means it must be shown *all* possible examples, and told which are positive examples and which are negative examples. For grammar induction the learner must be shown all possible finite strings. There are an infinite number of them, so this is an impossible task. (In fact, for grammar induction, it is easy to prove that there are infinitely many grammars consistent with any finite set of strings.)

The only way for the inducer to learn without receiving infinitely many examples is to bias the learner or to constrain the use of disjunction in the representation language. Goodman (1955) calls this the old riddle of induction: to learn anything at all, you either have to be biased or partially blind. By hypothesis, students do learn inductively. So it is only a question of whether they are biased, partially blind, or have some other way of solving the disjunction problem.

Prior solutions to the disjunction problem

Research in induction has used five major techniques for solving the disjunction problem. These will be reviewed briefly (see Cohen & Feigenbaum, 1983, for details). One technique has been mentioned already: disjunction-free induction. Winston's arch-learner was of this type (see section 2.6). Disjunctive generalizations are simply banned from the class of possible generalizations. This technique is the only one of the four that solves the problem by putting constraints on generalizations. (The remainder of this subsection is somewhat technical and can be skipped.)

The second "technique" is based on a celebrated theorem of Gold (1967). As it turns out, the technique is totally impractical in most cases because it requires that the inducer receive infinitely many examples. Gold's work dealt specifically with inducing grammars, but the results are more general. He proved three important theorems: (1) If a certain brute force inducer receives only positive examples, then it cannot learn the target generalization, except when the class of all possible generalizations is extremely restricted. (2) If the inducer receives both positive and negative examples, then it will eventually converge on the correct target generalization. (3) This brute force inducer is equivalent to all other inducers with respect to the theorems' results. Recently, certain psycholinguists have taken these results to mean that it will suffice to explain how language is learned if it can be shown that babies receive negative examples (see Pinker, 1979, for a review of this position). If they do, then they have all the information they need to induce their language, and moreover, it is pointless to inquire what kind of induction algorithm they might be using since all such algorithms are, in a certain sense, equivalent. This position chooses to ignore the fact that

Gold's second theorem requires that the inducer receive *all possible examples*, each bearing an indication of whether it is a positive example or a negative example. For any interesting formulation of inductive learning, this makes the example sequence be infinitely long. In concentrating on whether babies receive negative examples, the position ignores the physical impossibility of an infinite example sequence. Perhaps the infinite set of examples is taken as an idealization of a very large set, the set of all sentences that a baby hears while it is learning a language. However, the completeness of the example set is used crucially in the proof of Gold's theorems. It is easy to produce counterexamples to the theorem when the condition of completeness is violated. To put it differently, all that Gold really showed is that his brute force inducer converges on the correct generalization if and only if the example set is complete. The negative examples issue is a red herring. Given a finite example set, the brute force inducer may fail even if it does have negative examples. The only way to account for learning is to either (1) postulate strong restrictions on the class of all possible generalizations, as Winston did, or (2) to postulate a bias, as the remaining three techniques do.

The third technique uses a biasing measure based on extensions. For convenience in stating the bias, it allows only one top-level disjunction in its generalizations. That is, a generalization has the form `(OR` c_1 c_2... c_n`)` where the disjuncts c_i are disjunction-free generalizations. This is not a constraint on the class of possible generalizations. It does not decrease the expressive power of the class. It only puts the generalization in a form that makes it convenient to apply the bias measure. Bias is decided by comparing the *coverage* of individual disjuncts c_i: Given an example sequence, the coverage of a c_i is the set of examples in that sequence that it is consistent with. The coverage of a c_i is the intersection of its extension and the example sequence. Coverage is used to formalize biases. Various biases have been used reflecting varying assumptions about the induction task under study. Brown (1972) uses a bias that favors a generalization that has one c_i with as large a coverage as possible, along with an arbitrary number of c_i with small coverages. His induction task involves hypothesis formation over noisy data. The c_i with the largest coverage is the hypothesis. The other c_i cover the noise data. In other applications, the learner is biased to expect multiple hypotheses of about the same coverage (e.g., Vere, 1978; 1975; Hayes-Roth& McDermott, 1976). In this case, the bias is to take as few c_i as possible, each with the largest coverage possible. This bias implements one interpretation of Occam's razor.

The fourth technique is stochastic. The example set given to the learner has redundant examples. That is, an example may occur many times. The inducer's bias is based on finding a generalization that best fits the given example distribution. Generalizations are equipped with probabilities that are used to predict the example distribution. In particular, probabilities are assigned to disjuncts. Given a disjunctive concept, `(OR` c_1 c_2`)`, a probability P is assigned to c_1 while $1-P$ is assigned to c_2. The inducer's bias depends on a certain computation that calculates the likelihood of an example given a generalization that has probabilities assigned to its disjuncts. Then, for each generalization that the inducer constructs, probabilities are assigned to its disjuncts in such a way that the likelihood of the example distribution is maximized. The inducer then chooses the generalization that yields the maximal likelihood value. Thus, the inducer's bias is to choose generalizations that *best predict the distribution of examples.* A generalization's likelihood will depend on how many disjunctions the generalization has and where they occur. In general, the more disjunctions, the easier it is to fit the generalization to the examples, and hence the higher the likelihood value. A compensating bias is needed, otherwise the inducer will tend to generate trivially specific generalizations. Horning (1969) assigns prior probabilities to the generalizations in such a way that generalizations with more disjunctions (i.e., more degrees of freedom) have less prior probability.

A fifth major technique for solving the disjunction problem is to rate generalizations with some complexity measure (e.g., count the number of symbols needed to express it, as in Chomsky, 1975). The inducer is biased to choose the simplest generalization. This technique will often tend to overgeneralize, especially if there are few or no negative examples. For instance, a grammar inducer will tend to chose a grammar for the universal language (i.e., all possible finite strings) since universal grammars are often quite simple. Feldman (1972) balanced the complexity of the grammar itself against a second complexity measure, one based on the complexity of the derivation of the example strings from the grammar. For instance, one might measure a derivation's complexity by counting the number of parse nodes in the string's parse tree. Another technique is to balance the complexity-based bias, which tends to overgeneralize, against the likelihood-based bias, which tends to undergeneralize.

In addition to these five major techniques, there are many heuristic induction algorithms. It is often difficult to tell what their biases are. Consequently, they may have limited interest for theoretical psychologists. A heuristic inducer has been built for algebra equation solving, a domain presently under consideration. Neves' program, ALEX, induces procedures for solving algebra equations (Neves, 1981). ALEX's biases are woven into an algorithm for generalizing examples. ALEX will be discussed later as a representative for the class of heuristic approaches.

Competing hypotheses considered in this chapter

With this background in hand, it is time to consider how people solve the disjunction problem. Not all of the alternatives discussed above will be considered for this theory. The competition will be between the following five hypotheses:

1. No disjunctions: Disjunctions are not induced. Instead, they are there already, implicitly, in the set of primitive concepts that learners have when instruction begins. This solution to the disjunction problem is used by Winston (1975), Mitchell (1982), and others.

2. Domain-specific heuristics: Neves' ALEX program will be discussed as an example of learning by using ad hoc, domain specific biases.

3. Minimal disjuncts: The learner is biased to take generalizations with the fewest disjuncts possible. This is the essence of the solution used by Iba (1979), Michalski (1969, 1975), and others.

4. Exactly one disjunct per lesson: Given that the sequence of examples is partitioned into lessons, the learner acquires one new disjunct (subprocedure) per lesson.

5. At most one disjunct per lesson: Given that the sequence of examples is partitioned into lessons, the learner acquires at most one new disjunct (subprocedure) per lesson. That is, if the lesson doesn't require that the procedure be given a new disjunction, none will be installed.

The last alternative listed above is the one adopted by the theory. The first two competitors fall because they require implausibly strong assumptions about the students' states of knowledge prior to instruction. The fourth solution, introducing exactly one disjunction per lesson, makes bad predictions. The third solution, minimizing disjuncts, is empirically indistinguishable from the fifth solution, the one taken by the theory. However, the minimal-disjuncts hypothesis does not explain why lessons help instruction. It predicts, instead, that students would do just as well without the partitioning that lessons give the example sequence. They would learn identically from a sequence of examples chopped into hour-long slices at arbitrary places. Hence, the minimal-disjuncts hypothesis is rejected on grounds of explanatory adequacy: It does not explain why lesson structure has been found so universally helpful in education.

4.2 Barring disjunction from procedures

On one view, an inductive learning theory must either restrict the use of disjunction in the representation language or bias the inducer against making disjunctive generalizations. The former position is a little simpler. Despite the fact that it hasn't a prayer of explaining skill acquisition, it will be considered first because it provides an easy introduction to the tacit issues involved in controlling disjunctions.

By analogy with Winston's arch learner, it is easy to imagine a disjunction-free representation language for procedures Figure 4–3 shows a Winstonian representation for a worked subtraction problem. The representation is a semantic net. The three nodes labeled *a*, *b*, and *c* are the three visible writing actions of the example solution. The learner recognizes them as instances of the `DIFF` primitive, where `DIFF` is an action that takes the column difference and writes it in the column's answer place. Each of the three actions `ISA DIFF`. The representation uses an `AT` link to record which column the `DIFF` was taken in. The `NEXT` link represents the temporal sequence of the actions. The `LEFT` link represents the relative positions of the columns.

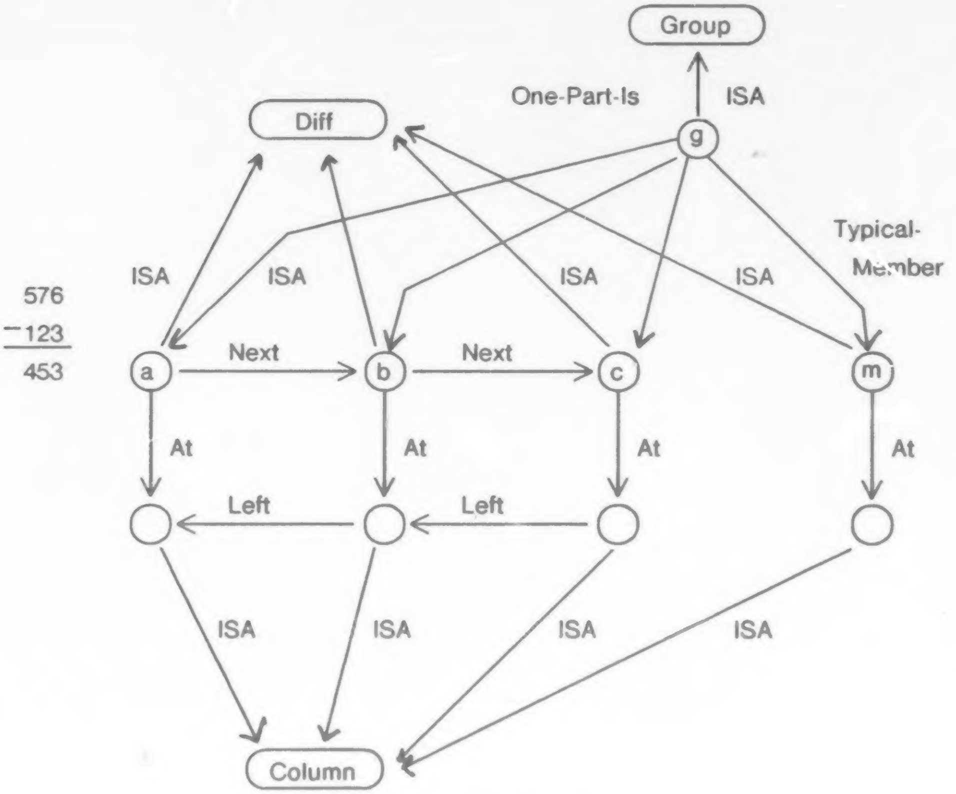


Figure 4–3
A worked subtraction exercise (on the left) represented as a semantic net.

Winston uses a special construction for representing groups of similar objects. In this example, the actions *a*, *b* and *c* are represented as a group. This is indicated by the fact that they are parts (via three `One-Part-Is` links) of the `GROUP` node, which is labeled *g*. The group has a typical member, labeled *m*, which `ISA DIFF`. Winston uses group nodes for iterative block structures, such as a column of arbitrarily many blocks. Here a group node is being used to represent a *loop*, a group of arbitrarily many column processing actions. Needless to say, more net structure than that shown in the figure would be necessary to do an adequate job of representing subtraction's main loop. However, this simple diagram gives enough detail to allow appreciation of the fundamental problem with this disjunction-barring approach.

The fundamental problem becomes apparent when a second subtraction problem is shown to the learner. Figure 4–4 illustrates the worked exercise and the semantic net for the generalization that the learner should produce. To induce the correct subtraction procedure (as many students do,

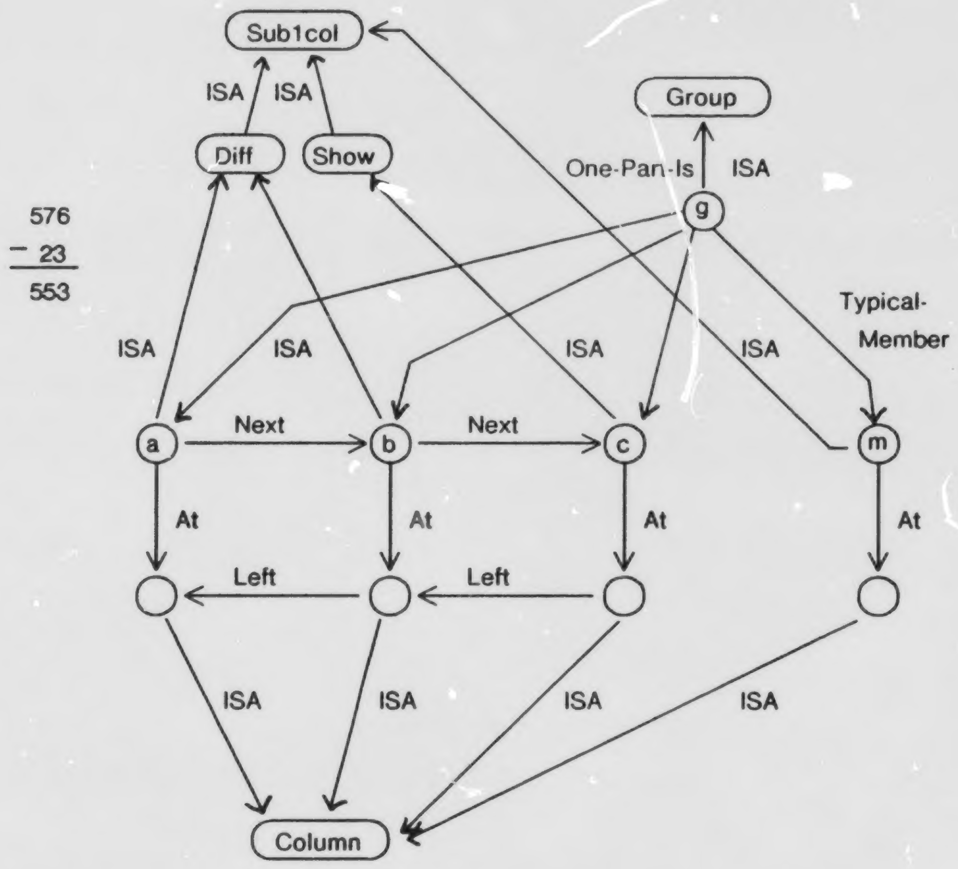

Sub1col
ISA
ISA
Diff
Show
Group
One-Part-Is
ISA
g
576
− 23
553
Typical-Member
ISA
ISA
ISA
ISA
a
Next
b
Next
c
m
At
At
At
At
Left
Left
ISA
ISA
ISA
ISA
Column

Figure 4–4

A worked subtraction exercise represented as a semantic net showing disjunctive column action.

so this is one criterion for adequacy), the learner has to recognize that the third action is not a `DIFF`, but another primitive action called `SHOW`. The `SHOW` action simply copies the top digit of a column into the answer. Given that the third action is not a `DIFF`, the typical member of the group (loop) cannot be a `DIFF`. It has to be the disjunction of `DIFF` and `SHOW`, which is represented by the node `SUB1COL`. This kind of generalization is just what the Winston's arch-learner does to induce that the lintel could be any `PRISM`. Prior to instruction, it thinks lintels were `BRICK`s. It is shown an arch with a `WEDGE` lintel. It induces that lintels could be any kind of `PRISM`.

However, there are major differences between the blocks world domain and the domain of written calculations. It is plausible that a child might know enough intuitive solid geometry to have a concept `PRISM` which disjoins `BRICK`, `WEDGE`, and a few other solids. However, it is rather implausible that a child has a concept `SUB1COL` that disjoins exactly the actions `DIFF` and `SHOW`. My intuition is that the closest a child would come to such a "naturally occurring" disjunction would be a concept, call it `DO1COL`, that disjoins four actions: `DIFF`, `SHOW`, taking the column sum, and writing the bottom digit of the column as the answer. It would take an experienced subtraction student to know that only `DIFF` and `SHOW` are members of subtraction's loop, and that the other actions are not. Without this experience, a student could only induce that any `DO1COL` is okay as a column processing action. This would generate bugs. The following problem illustrates the misconception that any `DO1COL` is okay as a column operation:

```
  8 4 5
- 1 2 1
  1 6 4
```

`DIFF` was used for the units column, the column sum was used for the tens, and the solver wrote the bottom digit as the answer for the hundreds column. Debuggy cannot diagnose non-deterministic bugs such as this one, hence no such bug occurs in the database. Nonetheless, it seems a plausible prediction. However, the disjunction-barring approach predicts that *all* students will have this bug, which is clearly false. The language cannot represent the correct procedure unless it has `SUB1COL`. It can't form `SUB1COL` with a condition because disjunction is banned. This approach can only fit the facts if `SUB1COL` is known by students before they take subtraction lessons. That is an absurd assumption. Barring disjunctions from the representation language fails.

It fails because it tends to overgeneralize. This is just what one would expect. It was shown earlier that when induction is allowed to generate arbitrarily many disjunctions, it undergeneralizes, yielding e.g., trivially specific generalizations. When induction is allowed to generate no disjunctions, it tends to overgeneralize. Describing human learning requires finding a middle way between free disjunction and disjunction barring. One way is to provide the learner with a set of "prefabricated" disjunctions, as Winston did. This is a nativist approach. However, such nativism is implausible for the calculation domain. It is absurd to assume that `SUB1COL` is innate, or even that it is learned prior to formal instruction in mathematics.

4.3 Neves' ALEX learner

Neves' program, ALEX, induces procedures for solving algebra equations (Neves, 1981). ALEX's biases are woven into an algorithm for generalizing an example. The algorithm has rules such as "all number constants are generalized by deleting the proposition which states which particular number the node is and just leaves the *isa* tag that says the node is a number. " (ibid, pg. 48). A tantalizing piece of the algorithm is "If the number of a term is in the condition then the sign of that term is also put in. There is no good rationale for this other than it works. The idea that the sign is important probably develops earlier on in the textbook." (ibid, pg. 48). If Neves' conjecture is right, then the student has acquired a constraint on generalization, an exciting instance of learning how to learn better. Neves' use of domain specific constraints in the clearly non-innate domain of algebra logically entails that some kind of constraint acquisition must have occurred. Unfortunately, he doesn't investigate the matter.

ALEX learns very quickly. It can acquire workably general knowledge of an algebraic transformation from a single example. The textbook Neves used to teach ALEX algebra was a high school textbook, but the lessons he chose concerned material that is often taught in primary school. It is quite likely that the lessons are actually review lessons. The lessons go through the material very quickly, much too quickly for Sierra, in fact. ALEX has such finely tuned biases for algebra induction that it is able to recover correct algebraic transformations even from these abbreviated review lessons. This leads to the conjecture that ALEX might make a good model for how people *relearn* material they used to know. Perhaps all they retain is induction biases. They use these to recover the procedure, whenever necessary, and forget the procedure itself.

4.4 Exactly one disjunct per lesson

A basic idea of step theory is to convert a difficult induction problem, induction with disjunctions, into a series of simple, disjunction-free induction problems. The easiest way to make this notion precise is to stipulate that there is exactly one disjunct acquired in each lesson. From the teacher's point of view, the sequence of examples is partitioned into lessons so that each lesson exemplifies one, and only one, new subprocedure (disjunct). From the learner's point of view, each lesson's examples are to be assimilated via disjunction-free induction. However, students have short attention spans and schools have schedules. A subprocedure might be too complicated to complete in the hour that is alotted to its lesson. So, it may be that some subprocedures must be taught in several lessons, in contradiction to the hypothesis. This possibility can be swiftly checked by examining the lessons sequences used in textbooks.

In most textbooks, there are lessons that are clearly intended to generalize a subprocedure taught earlier, rather than introduce a new one. For example, Houghton Mifflin's 1981 text introduces borrowing on two-column problems, such as *a*:

$$\text{a.}\ \begin{array}{r} \overset{5}{\cancel{6}}\,{}^{1}0 \\ -\ 2\ 3 \\ \hline 2\ 7 \end{array} \qquad \text{b.}\ \begin{array}{r} 7\ \overset{4}{\cancel{5}}\,{}^{1}2 \\ -\ 4\ 3\ 6 \\ \hline 3\ 1\ 6 \end{array} \qquad \text{c.}\ \begin{array}{r} \overset{5}{\cancel{6}}\,{}^{1}5\ 7 \\ -\ 2\ 6\ 3 \\ \hline 3\ 9\ 4 \end{array} \qquad \text{d.}\ \begin{array}{r} \overset{4}{\cancel{5}}\ {}^{1}\overset{4}{\cancel{5}}\,{}^{1}2 \\ -\ 3\ 6\ 7 \\ \hline 1\ 9\ 5 \end{array}$$

The next lesson uses examples, such as *b*, where all borrowing is from the tens column into the ones. Third lesson uses problems like *c*, where borrowing is in the left two columns. The fourth lesson teaches adjacent borrowing, as in *d*. Other textbooks use similar lesson sequence. McGraw-Hill's 1981 textbook series omits the *b* lesson. Scott-Foresman 1975 series compresses *b* and *c* into one lesson.

Under the hypothesis that there is exactly one new subprocedure per lesson, each of these lessons would start up a new borrowing subprocedure. Thus, the four lessons above would generate four borrowing procedures, one for each kind of problem. In particular, there would be distinct borrowing procedures for the units column and for the tens column. This would have several implications. When borrowing from zero is taught, it is always taught with three-column problems such as *e*:

e. $$\begin{array}{r} 7\ 9\ \\ 8^{1}0^{1}5 \\ -\,2\,1\,7 \\ \hline 5\,8\,8 \end{array}$$

f. $$\begin{array}{r} 7\ 9\ \\ 8^{1}0^{1}5 \\ -\,2\,1\,7 \\ \hline 8 \end{array}$$

g. $$\begin{array}{r} 7\ 9\ \\ 8^{1}0^{1}5 \\ -\,2\,9\,7 \\ \hline 8 \end{array}$$

This lesson would leave the borrowing procedure for the tens column totally unaffected. When the tens column is reached, the problem state is as shown in *f*. No borrowing is needed. Indeed, tens column borrowing is never needed when there is a borrow-from-zero (henceforth, BFZ) originating in the units column (see problem *g*). Consequently, the tens-column borrowing procedures are never invoked during the BFZ lesson. There is no reason to modify them. Leaving them alone makes a prediction that students will impasse if given problems that require borrowing from zero in the tens column, as *h* does:

h. $$\begin{array}{r} 7\ 9\ \\ 8^{1}0^{1}5\,4 \\ -\,2\,1\,7\,1 \\ \hline 5\,8\,8\,3 \end{array}$$

i. $$\begin{array}{r} 7\ \\ 8^{1}0^{1}5\,4 \\ -\,2\,1\,7\,1 \\ \hline 5\,2\,8\,3 \end{array}$$

Since the tens column borrow doesn't know how to BFZ, it will attempt to decrement the zero in the hundreds, violating a precondition, and reaching an impasse. Repairing this impasse would lead to bugs that have never been observed. Problem *i* shows the work of one such unobserved bug. I find such bugs implausible, but they are not star bugs.

However, *no* curriculum that I have seen teaches BFZ for the tens column in an explicit lesson. This means that *no* student will learn the correct procedure. They will all have bugs such as *i*. This is clearly a false prediction since many students eventually master subtraction. These students must have learned subtraction in ways not described by the theory (a situation that Occam's razor counsels us to avoid) or the original hypothesis that each lesson must start a new subprocedure is wrong.

To sum up: if there must be a new subprocedure per lesson, then there must be several distinct borrow subprocedures since several lessons are used in teaching simple borrowing. The correct algorithm requires that each be amended to handle borrowing from zero. Yet only one BFZ lesson occurs. Therefore, students should not be able to learn the correct procedure. Yet many do, so the hypothesis must be wrong.

4.5 Minimal-disjuncts vs. one-disjunct-per-lesson

This section discusses two solutions to the disjunction problem and contrasts them.

Always introducing a new subprocedure with a lesson has been shown to be empirically inadequate. A somewhat more complicated hypothesis is that the learner starts a new subprocedure for a lesson only if a new subprocedure is needed. If the lesson's worked example exercises can be handled by generalizing previously acquired material, then a new subprocedure is not added. This is the solution to the disjunction problem adopted by step theory. It is a felicity condition called one-disjunct-per-lesson.

Earlier it was shown that barring disjunctions from the representation language forced overgeneralization. This suggests a bias: allow disjunctions in the representation, but don't use a disjunction unless it is absolutely necessary. That is, the inducer prefers generalizations that have the minimal number of disjuncts. This will be called the minimal-disjuncts bias. This bias is a common technique in induction. The familiar arch-learning domain will be used to illustrate it.

Iba (1979) used a minimal-disjunction bias to solve the arch learning problem without prior knowledge of PRISM. He begins the induction by giving the learner two positive examples, the arch with the BRICK lintel and the arch with the WEDGE lintel. Since the inducer doesn't know the PRISM concept, it induces that the lintel can be any kind of block. It overgeneralizes. Iba then gives the inducer a negative example, a (pseudo-)arch with a pyramid as the lintel. This negative example is matched by the current arch concept. This means that the current arch is too general. It shouldn't match a negative example. Making the concept more general won't help of course. The only possible response is to form a disjunction. In this case, an appropriate disjunction would be to describe the lintel as (OR 'BRICK 'WEDGE). This illustrates what it means for an inducer to make the *fewest disjuncts possible.* The learner only inserts a disjunction when it has to. This minimal-disjuncts bias is one solution to the disjunction problem.

The main difference between the minimal-disjuncts bias and one-disjunct-per-lesson is that the minimal-disjuncts bias would detect the beginning of a new subprocedure even if it were in the middle of another subprocedure's lesson. That is, one-disjunct-per-lesson is a restriction on the minimal-disjunction hypothesis. Anything that one-disjunct-per-lesson can induce can also be induced by the minimal-disjunction bias, but not conversely. Hence, a critical case to look for is one where a subprocedure begins in the middle of another subprocedure's lesson. This would argue conclusively in favor of the minimal-disjuncts bias.

Leading zero suppression

I know of just one case where it could be argued that a disjunction must be started in the middle of a lesson. However, the evidence is rather unclear. It concerns leading zero suppression. Mastery of subtraction requires that the student suppress zeros in the answer if the zeros would be the leftmost digits. The answer to 58-50 is 8 not 08. This subskill is never given a lesson of its own in any of the textbooks that I've examined. Occasionally, the examples demonstrating another subskill (e.g., borrowing) will suppress a leading zero. But there are no lessons devoted solely to teaching the circumstances under which zeros should be left off the answer. Yet many students succeed in learning leading zero suppression. This would seem a rather clear piece of evidence against one-disjunct-per-lesson. However, the leading zero story is actually quite complex. Only the main points can be covered here.

If the minimal-disjuncts bias is to explain the acquisition of leading zero suppression, the textbooks would have to have a wide variety of leading-zero examples. The following examples illustrate the kind of variety needed:

$$\text{a.}\quad\begin{array}{rrr} & 6 & 7 \\ - & 6 & 3 \\ \hline & & 4 \end{array}\qquad \text{b.}\quad\begin{array}{rrrr} & & 4 & \\ & 7 & 5 & {}^{1}2 \\ - & 7 & 3 & 6 \\ \hline & & 1 & 6 \end{array}\qquad \text{c.}\quad\begin{array}{rrrr} & 0 & & \\ & 1 & {}^{1}5 & 7 \\ - & & 6 & 3 \\ \hline & & 9 & 4 \end{array}\qquad \text{d.}\quad\begin{array}{rrrr} & 4 & & \\ & 5 & {}^{1}5 & 7 \\ - & 4 & 6 & 2 \\ \hline & & 9 & 5 \end{array}$$

$$\text{e.}\quad\begin{array}{rrrr} & 7 & 5 & 9 \\ - & 7 & 5 & 6 \\ \hline & & & 3 \end{array}\qquad \text{f.}\quad\begin{array}{rrrr} & 0 & 9 & \\ & 1 & {}^{1}0 & {}^{1}7 \\ - & & 9 & 9 \\ \hline & & & 8 \end{array}\qquad \text{g.}\quad\begin{array}{rrrr} & & 4 & \\ & 5 & 5 & {}^{1}2 \\ - & 5 & 4 & 7 \\ \hline & & & 5 \end{array}$$

Under the minimal-disjuncts bias, it doesn't much matter where these examples occur, but they do have to occur somewhere. However, I have yet to see an example suppressing more than one zero used in any subtraction lesson. Examples of multiple zero suppression, such as *e, f* or *g,* do not appear. Despite their absence, some students acquire a complete understanding of leading zero suppression. Subtraction is the only columnar computation that generates answers with leading zeros: addition and multiplication do not. Long division often has subtraction problems that have multiple leading zeros, but only as a subproblem to the whole division problem. Perhaps these serve as examples for learning suppression of multiple leading zeros. However, many students are suppressing leading zeros before instruction in long division begins. They must have learned the skill some other way. In short, the evidence from the lesson data is not entirely clear. It doesn't clearly support the minimal-disjuncts bias since that hypothesis would have just as much trouble as one-disjunct-per-lesson in accounting for the acquisition of leading zero suppression.

Explaining why there are lessons

The strongest support for one-disjunct-per-lesson is that it explains why curricula are constructed they way they are. One-disjunct-per-lesson uses the lesson boundaries but the minimal-disjunction bias does not. A minimal-disjunction learner would learn equally well if the partitioning imposed by lessons were removed, leaving a continuous stream of examples and exercises. To the minimal-disjunction learner, lesson structure is irrelevant.

If lesson structure were irrelevant, then textbooks could be more simply laid out as a continuous stream of examples, exercises and other material. The teacher would use the daily math hour to get as far as possible through it. There would be no lesson boundaries. This is not how current (or past) textbooks are structured. Yet why have teachers adopted this lesson-structure format so universally? It can hardly be an accident or a fad. Teachers are dedicated and innovative enough that they would have dispensed with the straight jacket of lessons structure if they found it ineffective. To put it differently, if one accepts the nearly universal use of lessons as a natural phenomenon worth explaining, then one-disjunct-per-lesson explains it but the minimal-disjuncts bias does not. The one-disjunct-per-lesson hypothesis has greater explanatory adequacy than the minimal-disjuncts hypothesis.

The minimal-disjuncts bias predicts that students would learn equally well from a "scrambled" lesson sequence. To form a scrambled lesson sequence, all the examples in an existing lesson sequence are randomly ordered then chopped up into hour-long lessons.* Thus, the lesson boundaries fall at arbitrary points. The minimal-disjuncts bias predicts that the bugs that students acquire from a scrambled lesson sequence would be the same as the bugs they acquire from the unscrambled lesson sequence. This empirical prediction needs checking. If it is false, as I am sure it is, then the minimal-disjuncts bias can be rejected on empirical as well as explanatory grounds.

In short, we've arrived via a circuitous route at the felicity conditions thesis. It hold that teacher-student communication is a conversation of sorts that is governed by tacit conventions. The conventions facilitate learning. Perhaps it would be fun to close this discussion with a little speculation.

---

* Randomly ordering the examples would introduce a confounding effect. Examples from late lessons could appear before any of the examples of the preceding lesson. For instance, the subskill of borrowing-from-zero could be exemplified before borrowing-from-non-zero. Here is a scrambling without the confound: Suppose that the examples in the first lesson of the original sequence are labelled 1.1, 1.2, 1.3, etc. The examples of the second lesson are 2.1, 2.2, 2.3, etc. The other lessons' examples are similarly labelled. The scrambled lesson sequence is: 1.1, 2.1, 3.1, 4.1, etc. for the first lesson; then 1.2, 2.2, 3.2, 4.2, etc. for the second lesson, and so on. The scrambled lesson sequence introduces all of the procedures in the first lesson, then reviews it in each of the following lessons. If the minimal-disjuncts bias holds, this scrambled lesson sequence should yield the same bugs as the unscrambled lesson sequence.

Suppose one goes a step further than the felicity conditions thesis and conjectures that the felicity conditions that exist are those that *optimize* the information transmission. To solve the learner's disjunction problem, the teacher's optimal strategy would be to point to a node in the learner's knowledge structure and say "disjoin that node with the following subprocedure:...." Clearly, this is impossible. So the teacher says the next best thing, "Disjoin *some* node with the following subprocedure:...." The learner has to figure out which node to disjoin because the teacher can't point to it. But the learner knows now that some disjunction is necessary and that the examples following the teacher's command will determine its contents (this is the exactly-one-disjunct-per-lesson hypothesis that was discussed in section 4.4). If it were not for the exigencies of school scheduling, this would be perhaps the optimal information that felicity conditions could transmit. However, lessons have to be about an hour long. This means that only some of the lesson boundaries will correspond to the teacher's command to start a new disjunction. The other lessons will finish up the previous lesson. In short, the optimal feasible felicity condition for information transmission could well be the one-disjunct-per-lesson bias.

4.6 Formal hypotheses

The basic solution to the disjunction problem that people use has been uncovered. What remains is to express that hypothesis clearly and precisely. Three functions, named `Disjoin`, `Induce` and `Practice`, will be used to formalize one-disjunct-per-lesson. The functions will not be defined. Instead, they will gradually acquire meaning as the constraints of step theory are stated in term of them. The previously undefined function `Learn` will be defined in terms of them. The following is a list of the nomenclature, some of it duplicated from the previous chapter, with comments on their intending meanings.

`(Examples L)`	A function that returns the worked examples contained in its argument, which is a lesson.
`(Exercise L)`	A function that returns the practice exercises contained in its argument, which is a lesson.
`(Induce P XS)`	An undefined function that returns a set of procedures. Its first argument, `P`, is a procedure, and its second argument, `XS`, is a sequence of examples. It represents the various ways that its input procedure can be generalized to cover the examples. If there is no way to generalize the input procedure to cover the examples, `Induce` returns the null set.
`(Practice P XS)`	An undefined function that returns a set of procedures. Its first argument, `P`, is a procedure, and its second argument, `XS`, is a sequence of exercise problems. The output procedures correspond to the various ways that the input procedure can be generalized in order to solve the given problems.
`(Disjoin P XS)`	An undefined function that returns a set of procedures. Its first argument, `P`, is a procedure, and its second argument, `XS`, is a sequence of examples. It represents the insertion of a new subprocedure (disjunct) into the given procedure. Since there are sometimes several ways to do this, it returns several different procedures.

With these new terms in hand, the felicity condition can be formally stated. `P` and `L` stand for a procedure and a lesson:

One-disjunct-per-lesson

Let

```
(Learn P L) ≡
  If (Induce P (Examples L))≠{} then (Learn1 P L)
     else (Learn2 P L).
```

where

```
(Learn1 P L) ≡
  { P″ | ∃ P′ such that P′ ∈ (Induce P (Examples L))
                and P″ ∈ (Practice P′ (Exercises L)) }.
```

and

```
(Learn2 P L) ≡
  { P″ | ∃ P′ such that P′ ∈ (Disjoin P (Examples L))
                and P″ ∈ (Learn1 P′ L) }.
```

Moreover, `(Induce P XS)` and `(Practice P XS)` do not introduce into `P` any new disjunctions or any new disjuncts on old disjunctions, and `(Disjoin P XS)` inserts into `P` exactly one new disjunction or one new disjunct on an old disjunction.

The function `Learn` produces the set of procedures that can be acquired from a given lesson and a given initial procedure. If the procedure can be generalized without adding disjunctions, then no disjunction is introduced (the `Learn1` case). If there is no such generalization, then a disjunction is introduced and the resulting procedure is generalized (the `Learn2` case). `Learn` only introduces a disjunction if it has to. `Learn` is defined in terms of the three main undefined functions. Hence, it is just as undefined as they are. In fact, the last two clauses use the term "disjunction," which has not been formally defined. However, enough examples of disjunctions have been given that it should be clear what is meant even without a formal definition. The formal definition must await definition of the representation language used for procedures.

Chapter 5
The Invisible Objects Problem

The disjunction problem is perhaps the most famous problem in induction. A less well known but equally critical problem concerns what could be called invisible objects. An invisible object is something that is not present in an example given to the inducer, but is nonetheless relevant to the generalization being induced. In the domain of mathematical calculations, invisible objects are usually numbers. For instance, suppose the learner sees the teacher write 5 in *a*:

```
a.        4 7            b.        5 0
        - 1 2                    - 2 3
        -----                    -----
            5                        6
```

Consider two generalizations that explain the 5: The 5 is the difference of the digits in the units column, 7−2, or it is a more complicated combination of visible digits: (4+2)−1. The latter requires an invisible object, 6, the result of 4+2. Example *b* is consistent with the second generalization but not with the first. Its invisible object is 8.

The arch learning task provides another illustration of the invisible objects problem. One characteristic of an arch is that it has a gap right in the middle of it. The gap must be between the two legs, directly under the lintel, and directly on the supporting surface. As Winston points out (1975), one way to represent a gap is to use an invisible brick. Given this representational construct, an arch can be represented as

```
(AND  (ISA LINTEL 'PRISM)
      (ISA LEG1 'BRICK)
      (ISA LEG2 'BRICK)
      (ISA GAP 'BRICK)
      (INVISIBLE GAP)
      (SUPPORTS LEG1 LINTEL)
      (SUPPORTS GAP LINTEL)
      (SUPPORTS LEG2 LINTEL) ...)
```

The representation uses `(INVISIBLE GAP)` to indicate that the variable `GAP` is bound differently than the other variables when the pattern is matched. `GAP` can be bound only to "invisible bricks" while the other variables can be bound only to visible bricks. As it turns out, Winston does not use invisible object variables. His representation requires all variables to be bound to visible objects. The relationship `(NOT (TOUCHING LEG1 LEG2))` is used to express the gap between the arch's legs.

Although an explicit device, such as `INVISIBLE`, can be used to specify whether the objects bound to a variable are visible or not, a more common representation convention is to use functions to designate invisible objects and variables to designate visible ones. For instance, the arch's gap could be expressed using a distance function:

```
(> (DISTANCE/BETWEEN LEG1 LEG2) 0)
```

The output of the distance function is an invisible object, a number. The arch-concept states a constraint on this invisible object, that it be greater than zero. Functions can be used wherever variables can be used. Under this convention, the only difference between what a variable can designate and what a function can designate is that the variable's referent must be a visible object (c.f., Hempel's definition of confirmation, 1946). The syntactic distinction between function and variable replaces `INVISIBLE` as a way to control invisibility. Using functions to control invisibility is only a syntactic device. Any n-ary function can be converted to an (n+1)-ary relation, thereby allowing a variable to be bound to its output. Similarly, an (n+1)-ary relation can be converted to an n-ary, set-valued function. In principle, the representation has total freedom to control invisibility. Instead of `INVISIBLE`, it uses syntax. The net effect is the same.

The invisible object problem

Induction's troubles with invisibility come when the representation allows an expression to be expanded arbitrarily by adding constructions that designate invisible objects. Given an example, the learner can't see what invisible objects might be involved in the target generalization. The learner may make some educated guesses about which invisible objects are relevant, perhaps, then see if they play the same roles in the second example as they did in the first. Because the representation allows so many choices, the learner's problem of finding the relevant invisible objects is very hard (indeed, it will be shown later to be unsolvable). For instance, if Winston allowed invisible bricks, then they could be lying around anywhere. The learner would have no way to know if there were just one invisible brick, the gap, or dozens lying about all jumbled up. Similarly, if Winston allowed distance functions and the usual arithmetic functions, then the learner couldn't discriminate between

```
(> (DISTANCE/BETWEEN LEG1 LEG2) 0)
```

and

```
(> (ADD (DISTANCE/BETWEEN LEG1 LEG2)
        (DISTANCE/BETWEEN LEG1 LEG2))
   0)
```

The ADD function introduces a second invisible object, which is distinct from the one introduced by `DISTANCE/BETWEEN`. The learner has no way to know whether or not this new invisible object is worthy of description.

A better illustration of the invisible objects problem is provided by Langley's BACON3 program (Langley, 1979). It induces physical laws given tables of idealized experimental data. For instance, it can induce the general law for ideal gases when it is given "experiments" such as this one:

```
(AND  (MOLES 1.0)
      (TEMPERATURE 300.0)
      (PRESSURE 300000.0)
      (VOLUME 0.008320))
```

This formal representation describes the experiment in the same way that Winston's representation described a scene in the blocks world (this is not the representation that BACON3 uses, by the way). The expression above says that there is one mole of gas at a certain temperature and pressure, occupying a certain volume. The goal of BACON3 is to find a description that is a generalization of the experiments that it is given. For this series of experiments, the generalization that it induces is:

```
(AND   (MOLES N)
       (TEMPERATURE T)
       (PRESSURE P)
       (VOLUME V)
       (CONSTANT
            (QUOTIENT
                 (TIMES P V)
                 (TIMES N T))))
```

That is, `PV/NT` is a constant. This is one way to express the ideal gas law, which is more widely known as $pV = nRT$, where $R = 8.32$. In the representation above, notice that the last clause is a composition of functions that hides the intermediate results `PV` and `NT`. These intermediate results do not appear in the "scene" described earlier. This is what makes BACON3's job hard. BACON3's method for solving this induction problem is, very roughly speaking, to guess useful invisible objects descriptors and enter their values in the scenes. It might start by forming all binary function on the visible objects, e.g., `NT`, `P+V`, `N/N`, `PP`, `P/T`, etc. Since none of these yield values (invisible objects) that are constant across all the scenes, it trys further compositions: `NT/PV`, `NT+V`, `NTPV`, etc. At this level, it succeeds, since `PV/NT` turns out to be the same value, 8.32, in all the scenes. Essentially, BACON3 forms the simplest polynomial that is consistent with the scenes, where "simple" is defined computationally by the way that BACON3 organizes its search. Roughly speaking, it prefers the polynomial with the fewest intermediate terms (invisible object designators). It solves the invisible object problem by choosing a generalization with a minimal number of invisible object designators.

Four potential solutions to the invisibility problem will be discussed:

1. Banning invisibility: The knowledge representation language for mathematical procedures is defined so that no constructions designate invisible objects. This is the approach taken by Winston's `(NOT (TOUCHING LEG1 LEG2))` solution.

2. Unbiased induction: Enough examples are provided to the learner that all invisible object designators except the appropriate ones are eventually eliminated.

3. Minimal invisibility: The learner is biased to choose generalizations with the fewest invisible object designators (e.g., the fewest functions, if functions are what designate invisible objects). This is roughly what BACON3 does. (See also Brown's work on inducing kinship relations, 1972; 1973.)

4. Show work: First, the target concept is taught in such a way that all objects that would normally be invisible are somehow made visible. Then, the learner is re-taught the target concept, this time with the invisible objects invisible. The learner's task during the second lesson is only to discover which of the visible object designators that it already knows is now being used to designate an invisible object.

This chapter will take each hypothesis in order. The show-work hypothesis will be shown to engender the best empirical and explanatory adequacy.

5.1 Barring invisible objects

The simplest way to handle the invisible objects problem is to bar invisible object designators from the representation language. But this will not work for the domain of mathematical procedures. Constructions for designating invisible objects are needed so that one can represent procedures such as long column addition. Long column addition solves problems such as

$$\begin{array}{r} 3 \\ 4 \\ 5 \\ \underline{+6} \end{array}$$

The ordinary student solves this without jotting down intermediate results. The student keeps a running sum mentally. This requires some construction in the representation language that can designate invisible objects, namely the intermediate sums. Therefore, invisible objects cannot be barred from the representation language.

5.2 Unbiased induction with lots of examples

It is not hard to see that the invisible object problem is just as unsolvable as the disjunction problem. It is unsolvable in the sense that adding more examples doesn't narrow the set of consistent generalizations down to a singleton set. In some domains, one can even prove that it is unsolvable. Polynomial induction (e.g., BACON3) is a classic case that is particularly relevant to the domains addressed by this theory. Given a set of numbers pairs, $\{ \ldots\langle x_i, y_i\rangle\ldots\}$, the task is to induce a polynomial function such that $f(x_i)=y_i$ for all i. Such functions are generalizations of the set of example pairs. This induction task allows invisible objects in the representation. They are the intermediate results of the polynomials. A relational representation of the polynomial function $y=x^2+1$ is

```
(AND  (PAIR X Y)
      (TIMES Z X X)
      (INVISIBLE Z)
      (PLUS Y Z '1))
```

Here Z is used to designate x^2, the intermediate result of x^2+1. Since it does not appear in the example pair, it must be marked `INVISIBLE`.

If intermediate results (i.e., invisible numbers) were barred from generalizations (i.e., polynomial functions), then the problem of inducing polynomials from sets of pairs would be trivial. When invisible numbers are allowed, it is unsolvable. That is, given any finite set of pairs, there are infinitely many polynomial functions that generalize them. Proof: If there are n pairs, then there is always an $n-1$ degree polynomial that fits them. An n degree polynomial could fit the n pairs plus another pair, chosen randomly. Since there are an infinite number of possible extra pairs, there are an infinite number of n degree polynomials that will fit the n pairs. Q.E.D.

To pick an illustration closer to home, consider inducing the function nest that provides the answer to the tens columns in two column subtraction problems, such as:

a. $\begin{array}{r} 7\ 2 \\ -\ 4\ 1 \\ \hline 3\ 5 \end{array}$ b. $\begin{array}{r} 7\ 4 \\ -\ 2\ 1 \\ \hline 5\ 3 \end{array}$ c. $\begin{array}{r} 3\ 6 \\ -\ 1\ 2 \\ \hline 2\ 4 \end{array}$

Looking at *a* and *b*, an inducer might form the following generalizations:

1. $A_{10} = T_{10} - B_{10}$
2. $A_{10} = T_1 + B_1$
3. $A_{10} = ((T_{10} + T_1) - (B_{10} + B_1)) - A_1$

where the subscripts indicate the column, and T, B and A stand for the top, bottom and answer. The first generalization is the correct one. The second generalization is that the ten's answer is the sum of the units columns' digits. This second generalization, although consistent with examples *a* and *b*, is inconsistent with *c*. Many such accidental generalizations can be eliminated by giving lots of examples. However, generalization 3 cannot be eliminated. It will be true of any subtraction problem. This shows that there are some absurd generalizations, generalizations that students would never make, that would survive induction even over an infinite number of examples. Students must be applying other constraints to the induction process to eliminate this generalization, and many others like it.

5.3 Minimal number of invisible object designators

A close consideration of long column addition supports the idea that students might be biased to use as few invisible objects as possible. Students are introduced to long column addition with problems that have just three numbers to add. Given an example such as

$$\begin{array}{r} 3 \\ 4 \\ +1 \\ \hline 8 \end{array}$$

there are many ways to generate 8 from 3, 4 and 1. Each requires various intermediate results. Some possibilities are:

concept	*Number of intermediate results*
4+4	0
3+4+1	1
4x3 - 3 - 1	2
$4^2 - 3^2 + 1$	3

Most of these potential generalizations will be eliminated by other examples. However, there will always be many left, as shown in the preceding section. Unbiased induction will not tell the learner which generalization to learn. In particular, some students learn long column addition correctly, so they must be using some bias to choose among the many generalizations that are consistent with the examples. If the learner is biased to pick the generalization with the *fewest intermediate results*, the correct algorithm will be acquired.

There are many explanations one could give for why a learner might have such a bias. The generalizations with the fewest invisible objects are also the ones with the fewest number fact functions. It could be that the students are biased to choose short calculations because such calculations are the easiest ones to perform. On the other hand, the students could also be biased to reduce their short-term memory load: the generalizations with the fewest invisible objects are also the procedures requiring the least use of short-term memory. These explanations are plausible. Unfortunately, their predictions are indistinguishable from expressing the bias as a bias against invisible objects. The learning data provides no way to split them. Until other data are collected, it is a moot point whether the measure being minimized is fact function load, memory load, or invisible objects.

An experiment with the minimal-something bias

Sierra was originally implemented to have the bias just discussed. On the first example of a lesson, it would find all the fact function paths between the visible numbers of the example (subject to an ad hoc upper bound on path length). Each example after the first would remove paths that were inconsistent with its visible numbers. At the end of the lesson, Sierra would find and keep all the minimal length paths. These paths were (a) consistent with all the examples, and (b) of minimal length. These paths were the generalizations that Sierra generated as its predictions for what human learners would choose. Sierra was able to learn correct subtraction and many subtraction bugs using this bias. Ironically, long column addition, the procedure that provided the original motivation for inducing invisible objects, also proved to be its undoing.

Sierra's problem with long column addition was in forming the recursive loop that would allow it to solve problems with arbitrarily long columns. Given two-digit additions problems, it would form one action, roughly `(Write A (Add T B)))`. Given the next lesson, with triple-digit problems, Sierra would form a second subprocedure, yielding a new procedure that could be roughly expressed as

```
(If <triple-digit>
    then (Write A (Add  T (Add M B)))
    else (Write A (Add T B)))
```

where `T`, `M` and `B` refer to the top, middle and bottom digits of a triple-digit column. The clue that something is wrong is that Sierra did not use its knowledge of two-digit addition to help it learn three-digit addition. There is not use of the two-digit addition embedded in the triple-digit addition. Sierra developed the triple-digit function nest from scratch. However, because its bias was lenient about invisible objects, it had no difficulty inducing the nested `Add` functions. Given the next lesson, with four-digit columns, Sierra again added a new subprocedure, yielding a procedure that could be roughly expressed as

```
(If <four-digit>
    then (Write A (Add T (Add TM (Add BM B))))
elseIf <triple-digit>
    then (Write A (Add T (Add M B)))
    else (Write A (Add T B))
```

Sierra might have formed a recursion at this point, but it did not. Hence, the procedure it learned is unable do a five-digit column. But a human learner would, I expect, be able to solve a five-digit column after this much tutelage (I have no data on long column addition). The reason Sierra did not form the loop is that it couldn't recognize the three-digit problem hiding in the midst of the four-digit problem. Parsing of problems pays special attention to boundaries. The boundary-bias must be present in order for Sierra to generate several key subtraction bugs (see the discussion of Always-Borrow-Left, section 1.1). I think it would recognize the recursion given slightly longer columns, but this is difficult to test (the set of possible paths gets too big for the computer's address space when the paths are long). Since Sierra did not find the recursion until after four digit problems were presented, it became crucial to find out where in the lesson sequence human students are first expected to form the recursion: at three-digit, four-digit or five-digit problems? If Sierra was a good model of human learning, then human students would need longer problems than four-digit ones to learn the loop. A second grade textbook was purchased (something that should have been done long before). This led to the discovery of the show-work principle.

5.4 Show work

In almost all cases, textbooks do not require the student to do invisible object induction. Instead, whenever the text needs to introduce a subskill that has a mentally held intermediate result, it uses two lessons. The first introduces the subskill using special, ad hoc notations to indicate the intermediate results. Figures 5–1 and 5–2 show some examples. Since the intermediate results are written out in the first lesson, the students need guess no invisible objects in order to acquire the subskill. The learning of this lesson may proceed as if invisible object designators were banned from the representation language.

The second lesson teaches the subskill again, without writing the intermediate results. The second lesson is almost always headed by the key phrase, "Here is a shorter way to X" where X is the name of the skill. The students are being instructed that they will be doing exactly the same work (i.e., the same path of fact functions). They are left with the relatively simple problem of figuring out how the new material relates to the material they learned just the day before. This kind of learning might be called *optimization* learning. It is similar to induction. Indeed, I believe Sierra could be easily modified to handle optimization learning. However, subtraction curricula have no optimization lessons. (They would if teachers taught students to suppress scratch marks, but most do not these days.) Without instances of optimization learning, the bug data will not help in discovering what is the right way to formulate such learning. Optimization learning remains a topic for future investigation.

These considerations motivate the following hypothesis:

> *Show-work*
> In worked examples of a lesson, all objects mentioned by the new subprocedure are visible, unless the lesson is marked as an optimization lesson.

This hypothesis is not as formal as others, although its intended meaning is clear. Later, its formal impact will be built into the knowledge representation language. Essentially, functions will be prohibited in certain areas of the representation and strongly limited in others. The details, which depend on the representation's syntax, are deferred until section 15.1.

The show work hypothesis is quite clearly a felicity condition. Neither the teacher nor the student must obey it. Yet when they do, it is easier to transmit information. In Sierra, the combinatorics of collecting function nests can be almost entirely avoided. Presumably, human learners may also find learning easier.

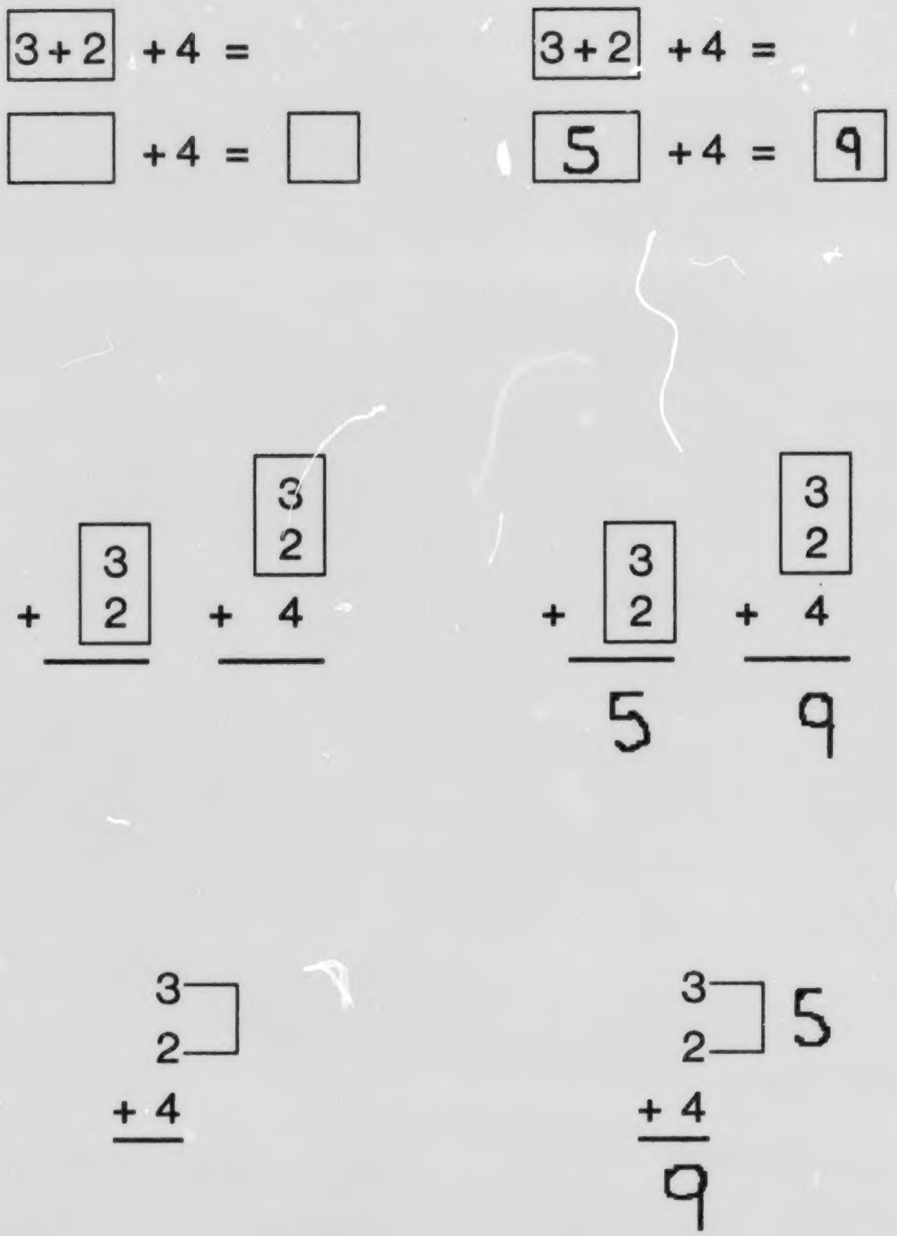

3+2 +4 =
+4 =
3+2 +4 =
5 +4 = 9
3
2
+ 4
+ 3
2
3
2
+ 4
+ 3
2
5 9
3
2
+ 4
3
2
+ 4
5
9

Figure 5–1
Three formats for column addition obeying the show work principle.
Exercises appear unsolved on the left, solved on the right.

$$\begin{array}{r} 23 \\ \underline{\times\ \ 6} \\ 138 \end{array}$$

$$\begin{array}{r} 23 \\ \underline{\times\ \ 6} \\ 18 \\ \underline{+120} \\ 138 \end{array}$$

$$\frac{6}{16} = \frac{3}{8}$$

$$\frac{6}{16} = \frac{6 \div 2}{16 \div 2} = \frac{3}{8}$$

tens	units
1	
2	9
+ 1	8
4	7

tens	units
2	9
+ 1	8
3	17

$\underline{47}$

Figure 5–2
Other exercise formats obeying the show work principle.
Exercises appear in normal format on left, in show-work format on right.

5.5 Invisible objects, disjunctions, and Occam's Razor

Occam's Razor is usually given a twofold interpretation. Webster's dictionary says Occam's Razor "is interpreted as requiring that the simplest of competing theories be preferred to the more complex or that explanations of unknown phenomena be sought first in terms of known quantities." If "simplest" means fewest disjunctions, then step theory claims that learners obey the first dictum of Occam's Razor. This was discussed in the preceding chapter. This chapter could be construed as showing that learners also obey the second dictum. For instance, if the learner seeks to explain where the teacher got the 6 from in the example

$$\begin{array}{r} 9 \\ -\ 3 \\ \hline 6 \end{array}$$

then Occam's Razor advises explaining it as 9-3, a function of known (i.e., visible) entities rather than some unknown (invisible) entity, e.g., the sum of numbers less than three, the student's age, the phase of the moon, etc. As Occam's Razor suggests, the invisible objects problem is a general problem, one that concerns almost any inductive account of knowledge acquisition. It's importance is highlighted by the apparent fact that teachers and learners have a special convention for solving it, the show-work felicity condition.

The invisible object problem and the disjunction problem are similar in many respects. Both can be solved trivially by barring their respective representational devices. This is not an option in this domain because mathematics procedures use both disjunctions and invisible objects. Both the invisible object problem and the disjunction problem are unsolvable by unbiased induction. If the class of all possible generalizations allows free use of them, then there are infinitely many generalizations consistent with any finite set of examples. Hence, both the disjunction problem and the invisible object problem require biased induction. In both cases, an empirically plausible bias is based on minimizing the uses of the respective devices (i.e., induction prefers generalizations with the fewest disjuncts and the fewest invisible object designators). However, these biases do not explain why lessons have the format that they do have. Better hypotheses are based on the idea of felicity conditions: conventions that make learning easier. The felicity condition hypotheses not only fit the facts, they also explain lesson formats as conventions for facilitating knowledge communication. They have the same observational adequacy as the minimization-based hypotheses, but they have more explanatory adequacy. They actually tell us something about why that mammoth cognitive-culture artifact — our educational system — has the properties that it does.

Chapter 6
Local Problem Solving

In the preceding chapters, the focus was on explaining learning. It was found that inductive learning could explain the gross features of student learning, provided that two felicity conditions were included in the explanation. However, there are two distinct but symbiotic foci of empirical curiousity to this investigation. Finding out how students learn is one; the other is finding out what causes them to have bugs. In this chapter and the next, the emphasis will be on explaining bugs. This chapter will introduce some bugs and bug migrations that will be referred to throughout this document.

Given the show work felicity condition and the one-disjunct-per-lesson felicity condition, inductive learning will converge. Given sufficient examples, an inducer will construct a large set of procedures. All the procedures will, by definition, be consistent with all the instructional examples. However, most will be buggy procedures instead of correct procedures. One cause is overgeneralization. An example of overgeneralization was described in section 2.7. Sierra's learner was given examples illustrating how to borrow from zero (henceforth, BFZ will be used to abbreviate "borrow from zero"). However, the learner overgeneralized the condition for executing the BFZ subprocedure, generating a bug that performs the BFZ subprocedure both for zero and for one (i.e., for identity elements). This is just one example of how learning can generate bugs.

This chapter argues that learning, and overgeneralization in particular, is a very powerful bug generator. It can, in principle, generate any conceivable bug. It is almost irrefutable. Constraints must be placed upon it if it is to have any explanatory value. But certain bugs are very difficult to generate if such constraints are placed on learning. In order to make explanatory, constrained learning empirically adequate, these bugs must be generated by another mechanism. The proposed generative source is local problem solving. To put it differently, this chapter begins by contrasting two positions:

1. an unconstrained learning theory, and
2. a constrained learning theory plus local problem solving.

Both can generate many bugs. However, when learning is used alone, it must be given so much flexibility that it can no longer explain why certain bugs are observed and not others. It has less explanatory adequacy.

6.1 Explaining bugs with overgeneralization

A simple learning framework bases explanations on overgeneralization. It explains errors as resulting from correct induction from impoverished sets of examples. For instance, the bug Diff-0−N=N, whose work appears in *a*:

a. $\begin{array}{r} 5\,0 \\ -\,1\,6 \\ \hline 4\,6 \end{array}$ b. $\begin{array}{r} 5\,6 \\ -\,1\,0 \\ \hline 4\,6 \end{array}$ c. $\begin{array}{r} \overset{4}{5}\,{}^{1}0 \\ -\,1\,6 \\ \hline 3\,4 \end{array}$

is explained as an overgeneralization of the correct rule N−0=N. The learner has seen examples such as *b* but not examples such as *c*. Hence, the learner induces that N−0=0−N=N, which is perfectly consistent with the instruction received so far. The overgeneralization framework is simple because it does not postulate "mislearning" as a source of errors. Instead, all learned concepts are consistent with the examples. Bugs arise only from overgeneralization, possibly in the context of incomplete instruction.

Simple overgeneralization is surprisingly powerful. The theorist can explain very diverse bugs by using it. It can even be used to explain bug migration, as Derek Sleeman has pointed out (Sleeman, submitted for publication). Before his suggestion is examined, an introduction to bug migration is in order.

Bug migration

Bugs are not usually stable. It is uncommon for a student to have exactly the same bugs on two tests, even if those tests are given only a day apart (see section 2.10). For instance, a student might have bugs A and B on Monday, but on Wednesday, the student has bugs A and C instead. Bug A was stable, but bug B was replaced by bug C. This is a kind of *inter*-test bug instability. Inter-test bug instability is the norm rather than the exception. Only 4% of the bugs remained stable in one study (VanLehn, 1981). Bunderson (1981) reports no stable bugs at all.

There is also *intra*-test instability. The bugs appear and disappear over the course of one testing session. A student may have bugs A and B on the first third of the test, bugs A and C on the second third, then just bug A on the last third.

One interesting kind of data is patterns of bug instability, and in particular, which bugs alternate with each other, as B and C did in the preceding illustrations. Many of the observed alterations will be spurious. B just happens to disappear at about the same time that C appears. They may have no interesting relationship to each other. However, some of the observed alternations seem highly significant. Not only do the bugs involved seem related intuitively, but the same groups of alternating bugs appear much more frequently than chance would predict. These significant alternations are termed *bug migrations*. A set of bugs that migrate into each other is called a *bug migration class*. Thus B migrates with C in the inter-test example above, hence {B, C} is a bug migration class. In the intra-test example, B alternated with C, but both bugs were absent on the last third of the test. It is quite common for bugs to migrate with a correct version of the procedure. To designate this, a null is used in the bug migration class: {B, C, Ø}.

As in the diagnosis of bugs, the diagnosis of bug migration requires careful analytical methods in order to guard against false positives: mistaken claims that a certain bug or bug migration class exists when in fact the cause of the observed behavior is just a chance alignment of unintentional errors (slips). Although the analytical methodology for bugs is quite highly developed, the equivalent technologicial development for bug migration has just begun.

Explaining bug migrations with overgeneralization

To return to Sleeman's point: certain cases of bug migration may be caused by overgeneralization. To explain an observed bug migration class, {B, C}, one postulates that the student has a generalized bug such that the ways to instantiate that generalized bug include the observed bugs B and C. For instance, earlier we saw that overgeneralization yields a rule:

> If there is a zero in the column, write the other digit in the answer.

which led ultimately to the bug Diff-0−N=N. A further generalization is the rule:

> If there is a zero in the column, write one of the digits in the answers.

This rule predicts an observed bug migration. The bug migration class contains two bugs. The first bug, Diff-0−N=0, solves problems as in *a*

```
a.      5 0          b.      5 0
      - 1 6                - 1 6
      -----                -----
        4 0                  4 6
```

It answers 0−N columns with zero. This bug results from instantiating the general rule by always taking the column's top digit for the answer. The second bug, Diff-0−N=N, whose work appears in *b*, results from instantiating the general rule another way, by always taking the column's bottom digit. The bug migration class is {Diff-0−N=0, Diff-0−N=N}. This bug migration is rather common. Figure 6–1 shows one student who exhibits it. On the first test, which was taken on a Monday, the student has two bugs Diff-0−N=0 and Borrow-Across-Zero. The later bug doesn't concern us here. (It affects problems *o, r, t* and *u*.) In all 0−N columns except one, the student answers with 0. In the exception column, problem *o*, the student did 0−N=N. This is a rather skewed example of intra-test bug migration. On the second test, taken two days later, the student still had the bug Borrow-Across-Zero, but now the student migrates freely between Diff-0−N=0 (problems *e, i, n, p, s, t,* and *u*) and Diff-0−N=N (problems *h, m, o,* and *q*). No instruction in subtraction was given between the two tests. The bug migration is apparently a product of some earlier experience. Overgeneralization offers one explanation.

The same generalized rule predicts a bug migration that involves exercises where there is a zero in the bottom of a column. For N−0 columns, either the top or the bottom digit is written as the answer. The migration is between Diff-N−0=N and Diff-N−0=0. The first instantiation, N−0=N, is correct. So this actually predicts an intermittent bug, i.e., the bug migration class is {Diff-N−0=0, Ø}. This bug migration has also been observed.

It seems that Sleeman's idea has some merit. Overgeneralization provides reasonable explanations for certain bug migrations as well as for the existence of certain bugs. The next section pushes farther.

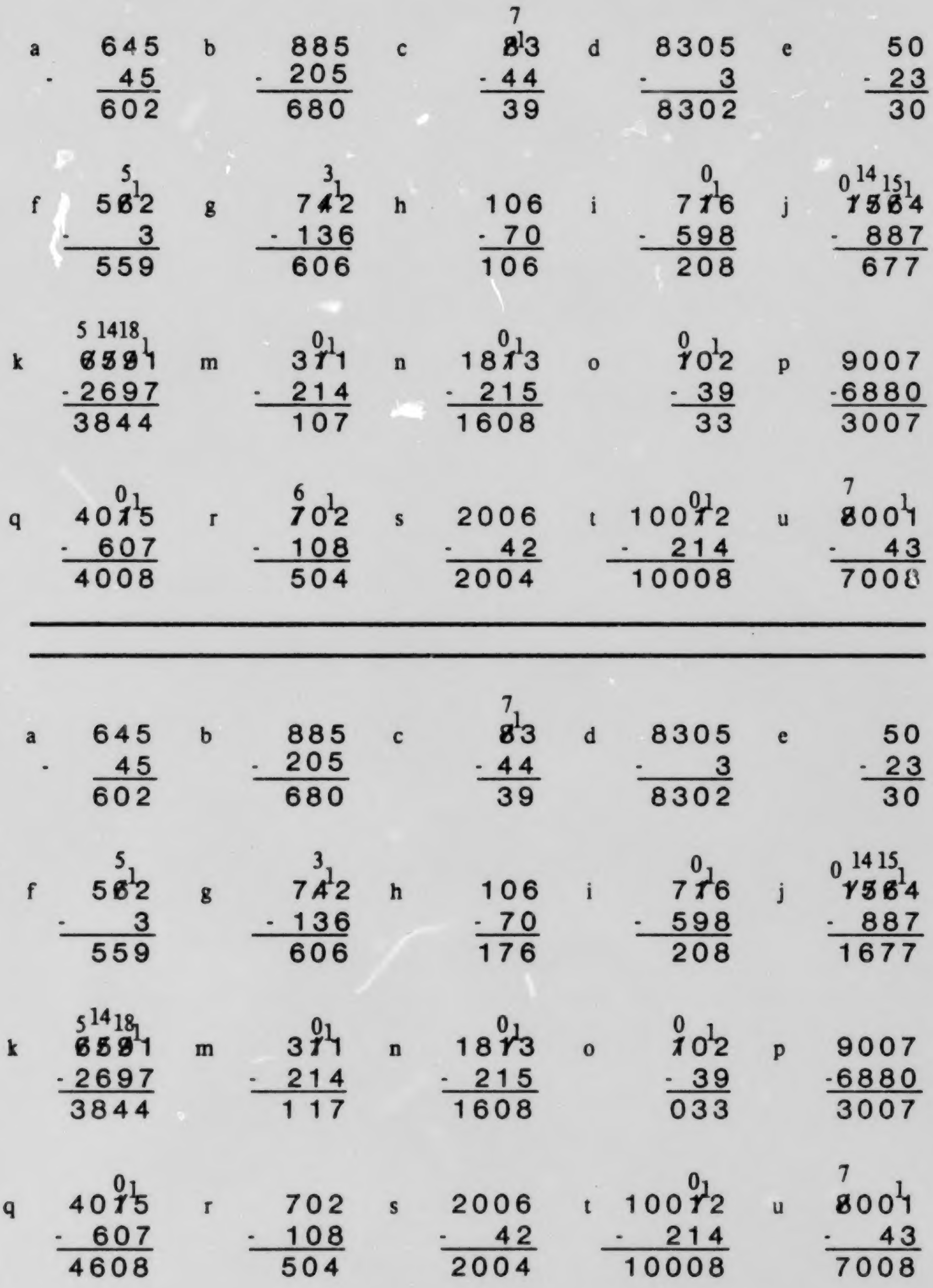

a 645 - 45 = 602
b 885 - 205 = 680
c 83 - 44 = 39
d 8305 - 3 = 8302
e 50 - 23 = 30
f 562 - 3 = 559
g 742 - 136 = 606
h 106 - 70 = 106
i 716 - 598 = 208
j 1564 - 887 = 677
k 6591 - 2697 = 3844
m 311 - 214 = 107
n 1813 - 215 = 1608
o 102 - 39 = 33
p 9007 - 6880 = 3007
q 4015 - 607 = 4008
r 702 - 108 = 504
s 2006 - 42 = 2004
t 10012 - 214 = 10008
u 8001 - 43 = 7008
a 645 - 45 = 602
b 885 - 205 = 680
c 83 - 44 = 39
d 8305 - 3 = 8302
e 50 - 23 = 30
f 562 - 3 = 559
g 742 - 136 = 606
h 106 - 70 = 176
i 716 - 598 = 208
j 1564 - 887 = 1677
k 6591 - 2697 = 3844
m 311 - 214 = 117
n 1813 - 215 = 1608
o 102 - 39 = 033
p 9007 - 6880 = 3007
q 4015 - 607 = 4608
r 702 - 108 = 504
s 2006 - 42 = 2004
t 10012 - 214 = 10008
u 8001 - 43 = 7008

Figure 6–1
Solution to two identical tests by student 1 of classroom 34.
First test is above the line, second test is below the line.

6.2 Stretching overgeneralization to account for certain bugs

Overgeneralization is a very powerful concept that can generate many bugs. On the other hand, the collection of observed bugs is very diverse. It is not clear whether the diversity of the bugs will defeat the generative power of overgeneralization. So far, overgeneralization has been applied in limited ways. The illustrations applied it to classes of numbers (i.e., borrow from *zero* was overgeneralized to borrow from *identity elements*) and to locations (i.e., zero in the top digit became zero *anywhere* in the column). This section leads off by discussing a bug migration class that requires overgeneralization to act in new ways. The bug migration class has three bugs:

{Borrow-Across-Zero,
Stops-Borrow-At-Zero,
Smaller-From-Larger-Instead-of-Borrow-From-Zero}

These bugs are each fairly important bugs in that they often occur in competitive arguments later in this document. They will be presented in some detail. An overgeneralization-based explanation will be given for each. The first bug's explanation is fairly smooth. The second is a little rougher. By the last one, overgeneralization will have been stretched to the breaking point.

The first bug in the class is Borrow-Across-Zero. This bug also cannot borrow from zeros. When it encounters a BFZ situation, it locates a nearby non-zero digit in the top row and decrements that instead. Figure 6–2 gives a problem state sequence illustrating it. The bug does its relocated decrement between states *a* and *b*. It does the rest of the problem correctly. The rather curious arrangement of decremented digits in the hundreds column is the hallmark of this bug.

To account for this bug with overgeneralization is not too hard. One postulates that the student has only seen non-zero borrowing. That is, the student has seen 52−19 and 511−99, but not 501−99. The student has induced that the "the digit to decrement is the closest top-row digit that is non-zero." This locative description is consistent with all the examples the student has received. It seems a little bit strange that the overgeneralization should mention zero despite the fact that the student has never seen a BFZ exercise. To justify its inclusion in the description, one would have to postulate that zero is so salient to the learner that its presence or absence is always recorded in generalizations. This is perhaps not implausible.

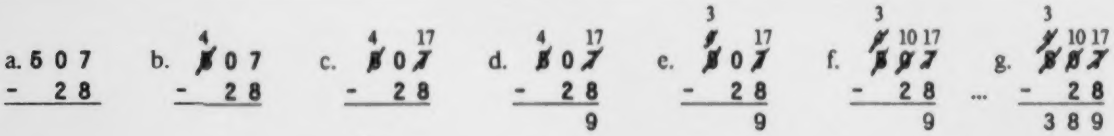


Figure 6–2
Problem state sequence for the bug Borrow-Across-Zero.

$$\text{a.}\begin{array}{rrr}5&0&7\\-&2&8\\\hline\end{array}\quad\text{b.}\begin{array}{rrr}5&0&\overset{17}{\cancel{7}}\\-&2&8\\\hline\end{array}\quad\text{c.}\begin{array}{rrr}5&0&\overset{17}{\cancel{7}}\\-&2&8\\\hline&&9\end{array}\quad\text{d.}\begin{array}{rrr}\overset{4}{\cancel{5}}&0&\overset{17}{\cancel{7}}\\-&2&8\\\hline&&9\end{array}\quad\text{e.}\begin{array}{rrr}\overset{4}{\cancel{5}}&\overset{10}{\cancel{0}}&\overset{17}{\cancel{7}}\\-&2&8\\\hline&&9\end{array}\quad\text{f.}\begin{array}{rrr}\overset{4}{\cancel{5}}&\overset{10}{\cancel{0}}&\overset{17}{\cancel{7}}\\-&2&8\\\hline&8&9\end{array}\quad\text{g.}\begin{array}{rrr}\overset{4}{\cancel{5}}&\overset{10}{\cancel{0}}&\overset{17}{\cancel{7}}\\-&2&8\\\hline3&8&9\end{array}$$

Figure 6–3
Problem state sequence for the bug Stops-Borrow-At-Zero.

The next bug in the bug migration class is Stops-Borrow-At-Zero (this bug was mentioned earlier, in section 2.9). When Stops-Borrow-At-Zero borrows from zero, it doesn't overwrite the zero, but skips the decrement operation entirely. Figure 6–3 shows a problem state sequence for this bug. The skipped borrow-from is evident at problem state *b*. The bug has already done the second step of borrowing, borrow-into. The rest of the solution is correct. The missing decrement is its only flaw. The missing decrement can be accounted for with overgeneralization by postulating that the student believes that "decrementing zero is null stuff." Perhaps the student justifies this by thinking, "If I have no apples and you try to take one, nothing happens." This generalized decrement operation accounts for Stops-Borrow-At-Zero.

A more difficult fact to explain is that these bugs migrate with Borrow-Across-Zero. In fact, the migration between Borrow-Across-Zero and Stops-Borrow-At-Zero is one of the most common bug migrations observed. To account for the migration, a generalization must be found that unifies the two generalizations:

1. Decrement the left adjacent, top-row digit, where decrementing zero is null stuff.
2. The digit to decrement is the first non-zero, top-row digit.

It would be simple just to disjoin these two generalizations. The student would believe that borrowing-from is either a null-stuff decrement or a decrement to a nearby digit. However, inducing disjunctive concepts is ruled out by the one-disjunct-per-lesson hypothesis. Hence, the disjoined concept must be present before instruction begins. A fairly exotic concept meaning "decrement zero is null stuff or nearby stuff" would have to be available during the induction of borrowing, perhaps by being in the base of primitive concepts.

The last bug in the bug migration class is Smaller-From-Larger-Instead-of-Borrow-From-Zero. Like the other bugs, it solves simple borrowing exercises correctly, but deviates from the correct algorithm when it is asked to borrow from zero. When a column requires a BFZ, the bug simply takes the absolute difference in that column, avoiding borrowing of any kind. (Figure 6–4 shows a problem state sequence.) The obvious explanation is that the student perceives BFZ as some kind of difficulty and avoids it by taking the absolute difference instead of borrowing. This makes sense if the student has not yet been taught BFZ and knows only how to do simple borrows. Note that this intuitively appealing explanation uses a problem solving framework. It postulates that the student detects problem situations and invents a way to avoid them. It falls outside the kinds of overgeneralization-based explanations that are currently being sought for this bug migration class.

$$\text{a.}\begin{array}{rrr}5&0&7\\-&2&8\\\hline\end{array}\quad\text{b.}\begin{array}{rrr}5&0&7\\-&2&8\\\hline&&1\end{array}\quad\text{c.}\begin{array}{rrr}\overset{4}{\cancel{5}}&0&7\\-&2&8\\\hline&&1\end{array}\quad\text{d.}\begin{array}{rrr}\overset{4}{\cancel{5}}&\overset{10}{\cancel{0}}&7\\-&2&8\\\hline&&1\end{array}\quad\text{e.}\begin{array}{rrr}\overset{4}{\cancel{5}}&\overset{10}{\cancel{0}}&7\\-&2&8\\\hline&8&1\end{array}\quad\text{f.}\begin{array}{rrr}\overset{4}{\cancel{5}}&\overset{10}{\cancel{0}}&7\\-&2&8\\\hline4&8&1\end{array}$$

Figure 6–4
Problem state sequence for the bug Smaller-From-Larger-Instead-of-Borrow-From-Zero.

To explain this bug as overgeneralization would be very difficult. One would have to postulate a way of viewing the borrow subprocedure as a whole since it is the whole subprocedure that is replaced by absolute difference. Presumably, such a viewpoint could be founded, but the theoretical costs of postulating such a large "primitive" would be high. Worse yet, this bug is in the same bug migration class as the other bugs mentioned above. The new "large primitive" would somehow have to generalize them as well.

The explanatory adequacy of overgeneralization

Essentially, these overgeneralization "accounts" are just building the observed bugs into the set of primitives that are assumed to be present before learning begins. A wide range of primitive concepts has been needed so far, just to capture three bugs. This would not be so bad if the primitives that generated bugs were somehow a natural class in that the class includes all concepts of a certain kind. But if all concepts that are "similar" to the ones needed so far (whatever that means) were allowed into the set of primitives, then the theory would overgenerate wildly. This abandons any chance of empirical adequacy. The opposite course is to drop the constraint that the set of primitives be somehow a natural class, and instead allow the theorist to dictate which primitives are in the set. This would improve the empirical adequacy, but it sacrifices explanatory adequacy. That is, the theory answers the question "why does this bug exist?" by saying "because this primitive concept exists," but it has no answer for the follow-up question, "Why does that primitive exist?" Such a theory doesn't explain the bugs, it only relabels them. It lacks explanatory adequacy.

6.3 Impasse-repair independence

The essential mechanisms of local problem solving are twofold: Problematic situations (called impasses) are detected, then they are solved or avoided (called repairing the impasses). At once one is struck by the apparent irrefutability of this framework. If the theorist is allowed to postulate anything as an impasse and a repair, then the theorist is allowed, in essence, to insert arbitrary condition-action rules into the procedure. The condition is the impasse and the action is the repair. It is clear that any conceivable bug could be generated this way, by inserting the appropriate condition-action rule *qua* impasse-repair combination. Such a framework would have no explanatory value. If one asked it why a certain bug existed, it would answer only "because a certain impasse-repair combination happens to exist."

The stipulation of an impasse and a repair would have some explanatory force if one could provide *independent evidence* for the impasse and for the repair. That is, stipulating an impasse I_1 and a repair R_1 to explain a certain bug would be believable if one could also exhibit a second bug generated by repairing the impasse I_1 with another repair, R_2. This would be independent evidence for the stipulated impasse I_1. Similarly, a good explanation requires independent evidence for the repair, such as a bug that results from using the same repair to a different impasse, called it I_2. That is, to explain the original bug, one needs to produce the arrangement of evidence shown in this table:

	I_1	I_2
R_1	Bug	Bug′
R_2	Bug″	

The original bug to be explained is Bug. Bug′ justifies the stipulated repair and Bug″ justifies the stipulated impasse. Actually, if the goal is to ascertain whether the local problem solving framework is correct, then it seems that requiring the existence of a fourth bug, the combination of I_2 with R_2 is necessary. The essence of problem solving is that *any solution that works is acceptable.* If there are several possible means to an end, and if problem solving is truly the activity going on, then each means will eventually be applied by someone to achieve the goal (all other things being equal). In this case, the goal (end) is to be in a non-impasse state. Hence, the bare notion of problem solving predicts that all repairs will be applied, by someone at some time, to each impasse. If a certain impasse-repair combination predicts a star bug, then the theory should provide an explanation for why that repair was not a reasonable choice for solving the problem presented by the impasse. If it could not, then one would begin to suspect that the framework wasn't really problem solving but something else instead. In short, the independence of impasses and repairs is a crucial, defining principle of local problem solving. The set of predicted bugs is exactly the set of all repairs applied to all impasses. Any exceptions must be explained by the theory. Put differently, *Bugs* = *Impasses* × *Repairs*, where × stands for the Cartesian product of two sets.

A Cartesian product bug pattern

The bug data have many instances of the kind of Cartesian product pattern that local problem solving predicts. This will be illustrated with the three bugs mentioned earlier, paired with three new bugs. The rest of this section presents this pattern. It and others like it are prime evidence for the local problem solving framework. This Cartesian product pattern has two impasses and three repairs:

	decrement zero	larger from smaller
Noop	Stops-Borrow-At-Zero	Blank-Instead-of-Borrow
Refocus	Borrow-Across-Zero	Smaller-From-Larger
Backup	Smaller-From-Larger-Instead of-Borrow-From-Zero	Doesn't-Borrow

Bugs from the same impasse are in the same column. Repairs label the rows of the bugs they generate. The bugs will be discussed row by row.

Stops-Borrow-At-Zero:

$$\begin{array}{r} 3\;4\;5 \\ -\,1\;0\;2 \\ \hline 2\;4\;3 \end{array}\ \checkmark \qquad \begin{array}{r} \overset{3}{} \\ 3\;4^{1}5 \\ -\,1\;2\;9 \\ \hline 2\;1\;6 \end{array}\ \checkmark \qquad \begin{array}{r} 1\phantom{^{1}0^{1}7} \\ 2^{1}0^{1}7 \\ -\,1\;6\;9 \\ \hline 4\;8 \end{array}\ \times$$

Blank-Instead-of-Borrow:

$$\begin{array}{r} 3\;4\;5 \\ -\,1\;0\;2 \\ \hline 2\;4\;3 \end{array}\ \checkmark \qquad \begin{array}{r} 3\;4\;5 \\ -\,1\;2\;9 \\ \hline 2\;2\; \end{array}\ \times \qquad \begin{array}{r} 2\;0\;7 \\ -\,1\;6\;9 \\ \hline 1\;\; \end{array}\ \times$$

Correctly answered problems are marked with √, and incorrectly answered problems with ×. The first bug, Stops-Borrow-At-Zero, is generated by assuming that the student has not been taught how to borrow from zero. When the student tries to use simple borrowing on BFZ problems, such as the third problem, an attempt is made to decrement the zero. The student recognizes that zero cannot be decremented. An impasse occurs. The student has detected a local problem that needs to be solved before any more of the procedure can be executed. The repair, called Noop (pronounced "no op"), simply causes the student to skip the stuck decrement action (i.e., it turns the action into a "null operation" or "no-op" in computer jargon). This leads to Stops-Borrow-At-Zero shown above (see figure 6–3 for a problem state sequence illustrating its solution).

The second bug is Blank-Instead-of-Borrow. Superficially, it looks very different from Stops-Borrow-At-Zero. It doesn't do any borrowing, but instead leaves unanswered just those columns that require borrowing. The explanation for this bug assumes that the student hasn't learned how to borrow yet. When the student attempts to take a larger number from a smaller one, an impasse occurs, presumably because the student knows that "you can't take a big number from a small one." The repair to this impasse is the Noop repair. It causes the column difference action to be skipped. This explains why borrow columns have blank answers. In general, the Noop repair causes actions that are "stuck" to be skipped. It is perhaps the easiest of all possible repairs. It is a quite straightforward solution to the problem of being unable to execute an action.

Borrow-Across-Zero:

$$\begin{array}{r} 3\,4\,5 \\ -\,1\,0\,2 \\ \hline 2\,4\,3 \end{array}\ \checkmark \qquad \begin{array}{r} 3\,\overset{3}{4}\,{}^{1}5 \\ -\,1\,2\,9 \\ \hline 2\,1\,6 \end{array}\ \checkmark \qquad \begin{array}{r} \overset{\overset{0}{1}}{2}\,{}^{1}0\,{}^{1}7 \\ -\quad 6\,9 \\ \hline 4\,8 \end{array}\ \times$$

Smaller-From-Larger:

$$\begin{array}{r} 3\,4\,5 \\ -\,1\,0\,2 \\ \hline 2\,4\,3 \end{array}\ \checkmark \qquad \begin{array}{r} 3\,4\,5 \\ -\,1\,2\,9 \\ \hline 2\,2\,4 \end{array}\ \times \qquad \begin{array}{r} 2\,0\,7 \\ -\,1\,6\,9 \\ \hline 1\,6\,2 \end{array}\ \times$$

Borrow-Across-Zero is generated by applying the Refocus repair to the decrement-zero impasse. The basic idea of the Refocus repair is to shift the external focus of attention, in this case, where to perform the decrement operation. Refocus shifts focus in a way that maintains some faithfulness to the procedure's description. As before, the assumptions are that the student knows that zero can't be decremented but does not know how to borrow from zero. The procedure that the student is following presumably describes the place to decrement as the top digit in the column just left of the column currently being processed. Refocus relaxes that description somewhat, shifting focus to the top digit in *some* column left of the current column. Any column that will allow the decrement operation to succeed is a potential candidate. In this case, only the hundreds column qualifies, so it is chosen. (Figure 6–2 gives the problem state sequence of the bug's solution.)

Smaller-From-Larger answers columns that require borrowing with a number that is the absolute difference of the two numbers. There are several ways to explain this bug. Here, the assumption is that the student reaches an impasse because he must process a column where the bottom digit is too large, and he understands that one can't take a larger digit from a smaller one. The Refocus repair relaxes the description of the arguments to the column difference operation. It relaxes the constraint on relative vertical positions. The operation is performed as if the column were inverted. This allows it to answer the column, thus coping with the impasse.

Smaller-From-Larger-Insteadof-Borrow-From-Zero:

$$\begin{array}{r} 3\,4\,5 \\ -\,1\,0\,2 \\ \hline 2\,4\,3 \end{array}\ \checkmark \qquad \begin{array}{r} 3\,\overset{3}{4}\,{}^{1}5 \\ -\,1\,2\,9 \\ \hline 2\,1\,6 \end{array}\ \checkmark \qquad \begin{array}{r} \overset{2}{3}\,{}^{1}0\,5 \\ -\,1\,6\,7 \\ \hline 1\,4\,2 \end{array}\ \times$$

Doesn't-Borrow:

$$\begin{array}{r} 3\,4\,5 \\ -\,1\,0\,2 \\ \hline 2\,4\,3 \end{array}\ \checkmark \qquad \begin{array}{r} 3\,4\,5 \\ -\,1\,2\,9 \\ \hline \end{array}\ \times \qquad \begin{array}{r} 2\,0\,7 \\ -\,1\,6\,9 \\ \hline \end{array}\ \times$$

These two bugs illustrate the Backup repair. Backup is perhaps one of the most difficult repairs to present, although it underlyingly quite simple. The essence of the Backup repair is retreating in order to take another alternative path. Backup resets the execution state of the interpreter back to a previous decision point in such a way that when interpretation continues, it will choose a different alternative than the one that led to the impasse that Backup repaired. In most cases, using Backup causes a secondary impasse. This is just what happens with Smaller-From-Larger-Instead-of-Borrow-From-Zero. As with the other bugs, the student reaches an impasse trying to decrement the zero in the tens column. The Backup repair gets past the decrement-zero impasse by "backing up," in the problem solving sense, to the last decision which has some alternatives open. After the repair, the student tries to process the units column in the ordinary way. Immediately he hits a second impasse, since he knows that one can't take a larger number from a smaller one. This second impasse is repaired by Refocus, yielding absolute difference as the answer in the units column. The student finishes up the rest of the problem without difficulty. The derivation of this bug is a little complicated. One should perhaps just try to get a rough sense of it now. Later, it will be presented in detail.

The bug Doesn't-Borrow is simple. Whenever it encounters a column that requires borrowing, it gives up on doing the rest of the problem, and goes on to the next problem on the test, if there is one. The bug is generated by applying the Backup repair to the impasse of being unable to take a column difference. At this point in the procedure, the most recent decision is not the decision about borrowing, because the student doesn't know about borrowing yet. Instead, the most recent decision involves whether to do the problem at all. The Backup repair retreats to this decision, and takes the open alternative: the student gives up on this problem, and goes on to the next.

The repair-impasse independence principle makes predictions

A crucial fact about the repair process comes out clearly in the Cartesian product pattern. It is the *independence* of repairs and impasses. Every repair is applicable to every impasse. In principle, a bug will be found for each pairing of an applicable repair with an impasse.

Of course, some repairs are much more popular than others, and some impasses are more common than others. Combining an unpopular repair with an uncommon core procedure may predict a bug that has not yet been observed. In fact, several bugs have been predicted by repair-impasse independence, then observed later. When the original model for repair theory was first tested, in September 1979, it predicted 16 bugs that had not yet been observed. When its predictions were test against newly collected data in December 1979, 6 of the predicted bugs were discovered (Brown & VanLehn, 1980). Since then, another of the original model's predicted bugs has been discovered even though few new data have been acquired in the interim. So, one of the chief advantages of the impasse-repair independence principle is that it makes predictions that can be used to focus empirical investigations and to test the theory.

Repair-impasse independence vs. bug occurrence statistics

It is not the case that repairs and impasses are statistically independent. Although rare bugs result from using uncommon repairs to uncommon impasses, it is not always the case that combining a common repair and a common impasse results in a common bug. The frequencies of the six bugs discussed above show this effect:

	decrement zero	larger from smaller
Noop	64 Stops-Borrow-At-Zero	1 Blank-Instead-of-Borrow
Refocus	44 Borrow-Across-Zero	115 Smaller-From-Larger
Backup	5 Smaller-From-Larger-Instead of-Borrow-From-Zero	0 Doesn't-Borrow

These figures show the number of students in a sample of 1147 who had the specified bug (see appendix 3). In this sample, the two impasses are equally common. 110 students had the decrement-zero impasse, and 116 students had the larger-from-smaller impasse. However, there is a strong skew in repair preferences. The Noop and Refocus repairs were equally popular for the decrement-zero impasse, but the Refocus repair was strongly preferred by students who can't borrow. This shows that a *simple* assumption of statistical independence is quite unwarranted. Repair-impasse independence does not mean statistical independence.

However, there are several problems with bug frequency data. If these can be solved, statistical independence may be found. The main problem is that most bugs are rather uncommon, occurring less than a half dozen times even in large samples (see appendix 3). This makes statistical inferences unreliable. A more subtle difficulty is that many bugs have multiple causes. Multiple derivations make bugs more common than simple frequency models would predict. For instance, Smaller-From-Larger is common because it has at least two derivations — one as the application of the Refocus repair and another as overgeneralization. The overgeneralization account is simply that the learner chooses absolute difference as the generalization of examples such as

$$\begin{array}{r} 5 \\ \underline{-2} \end{array}$$

On this account, students believe that $5-2=2-5=3$ despite the fact that they have never seen examples of the latter case, $2-5=3$. Given this concept of column difference, the students solve borrow columns, e.g., the units column of

$$\begin{array}{r} 42 \\ \underline{-15} \\ 33 \end{array}$$

without reaching an impasse. Thus, Smaller-From-Larger has a derivation as induction from an impoverished set of examples. Accounting for bug frequencies would have to take such multiple derivations into account.

Summary

The main purposes of repair-impasse independence are (1) to capture an important trend in the data, the Cartesian product pattern, and (2) to give a rigorous expression of the basic notion of local problem solving as multiple means to the same end, and (3) to rescue the theory from the irrefutability of allowing the theorist to postulate arbitrary, non-independent impasse-repair pairs. To the extent that the pattern holds across the data, the local problem solving framework is vindicated. The local problem solving explanation loses its force if independence has too many exceptions. To put it differently, the principle sets the default to independence. Any time a particular repair-impasse combination leads to a star bug, the theory must explain why.

6.4 Dynamic vs. static local problem solving

In the AI literature, the basic idea of detecting problems in a procedure and fixing them is not new. Sussman's HACKER program had two kinds of problem detection and rectification systems (Sussman, 1976). One acted dynamically, that is, during the execution of the procedure. It would detect problems such as trying to place a physical object in a space occupied by another object. The second system acted statically: it would examine the procedure as a goal-subgoal hierarchy, looking for patterns of conflicting goals. It could thus detect some problems without ever running the procedure. The same choice exists for local problem solving in this domain: impasses can be detected and repaired dynamically or statically. To put it intuitively, the issue is *when* the local problem solving process is carried out by the student. It could be that the local problem solving process is something like forgetting or mislearning. It could happen while the student is sleeping, or watching the teacher, or explaining the procedure to a friend. All that one can see in the Cartesian product pattern is the result of repair, and not *when* it happened. This section is a competitive argument between two approaches. The two hypotheses are (LPS will be use to abbreviate "local problem solver"):

1. Dynamic LPS: impasses are detected during the execution of the procedure. Repairs are made to the current state; the procedure itself is not modified.
2. Static LPS: impasses are detected by analyzing the structure of the procedure without executing it. Repairs are made by changing the procedure's structure.

Really, there is hardly any controversy (that is why the argument has not been given a chapter of its own). The evidence is clearly on the side of dynamic LPS.

Bug migration and stable long-term bugs

Intuitively, bug migration is a strong argument for the dynamic local problem solving hypothesis. But as it turns out, the static LPS hypothesis can do just as well at predicting bug migration although it must be accompanied by a simple (and ad hoc) ancillary assumption.

As discussed earlier in this chapter, bug migration is the phenomenon of a student switching among two or more bugs during a short period of time with no intervening instruction. The bugs the student switches among are called a bug migration class. The theory aims to predict which sets of bugs will occur as bug migration classes. With regard to local problem solving, the basic idea is that the bugs in a bug migration class result from *applying different repairs to the same impasse.* That is, the student appears to have the same procedure throughout the period of observation, but chooses to repair its impasses differently at different times. This basic idea is independent of whether local problem solving takes places statically or dynamically. Figure 6–5 presents an example of bug migration among several of the bugs discussed earlier. It illustrates how several bugs can occur on the same test by application of different repairs to the same impasse.

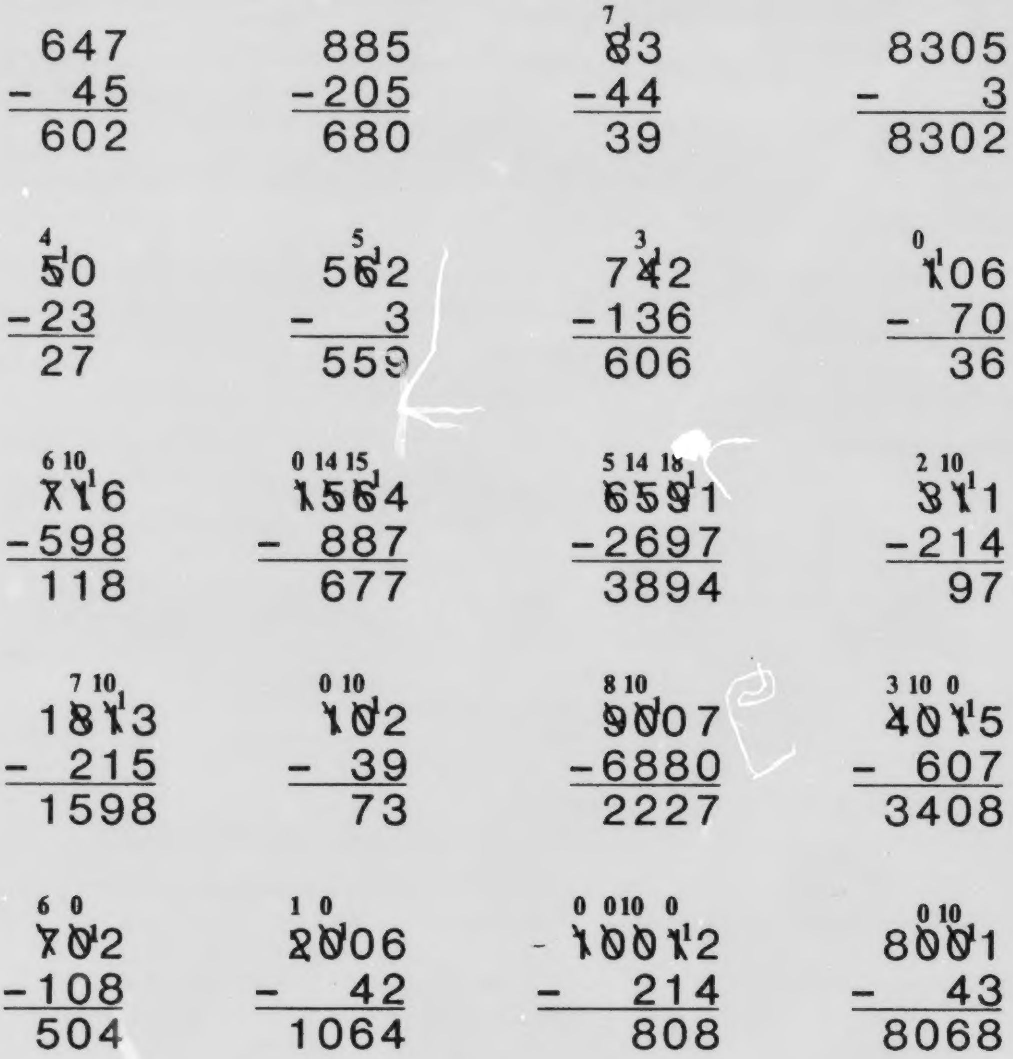

647 885 83 8305
- 45 -205 -44 - 3
602 680 39 8302
4 5 3 0
50 562 742 106
-23 - 3 -136 - 70
27 559 606 36
6 10 0 14 15 5 14 18 2 10
716 1564 6591 311
-598 - 887 -2697 -214
118 677 3894 97
7 10 0 10 8 10 3 10 0
1813 102 9007 4015
- 215 - 39 -6880 - 607
1598 73 2227 3408
6 0 1 0 0 010 0 0 10
702 2006 10012 8001
-108 - 42 - 214 - 43
504 1064 808 8068

Figure 6–5
Solution to a test by student 22 of classroom 34
showing intra-test bug migration.

Figure 6–5 is an exact reproduction of a test taken by student 22 of class 34. She misses only six problems, namely the ones that require borrowing from zero. The first two problems she misses (the second and third problems on the fourth row) are answered as if she had the bug Stops-Borrow-At-Zero. That is, she gets stuck when she attempts to decrement a zero, and uses the Noop repair in order to skip the decrement operation. The next two problems she misses (the first two problems on the last row) are answered as if she had the bug Borrow-Across-Zero. She hits the same impasse, but repairs by relocating the decrement leftward using the Refocus repair. On the third problem of the last row, she uses two repairs within the same problem. For the borrow originating in the tens column, she uses Backup to retreat from the decrement-zero impasse. She winds up writing a zero as the answer in the tens column (as if she had the bug Zero-Instead-of-Borrow-From-Zero). In the hundreds column, she takes the same Refocus repair that she used on the preceding two problems. On the last problem, she uses the Noop repair for both borrows.

Patches present a problem for the dynamic LPS hypothesis

The student of figure 6–5 is typical in that her repairs occur in runs. The first two repairs are one kind, the next two are another, and so on. This observation suggests that there can be a temporary association of an impasse with a repair. These pairs are called *patches.* Apparently, the first time the student of figure 6–5 hit the impasse, she searched for an applicable repair and not only used it, but created a patch to remember that she used it. On the next problem, she again encounters the impasse, but instead of searching for a new repair, she just retrieves the patch, and uses its repair. She completes the next problem without encountering the impasse, which is apparently enough to cause her to forget her patch, since the next time she hits the impasse, she repairs it a new way. Either the patch was forgotten during the non-impasse problem, or she chose to ignore it and try a different repair. The latter possibility is supported by her behavior at the end of the test, where she is applying different repairs for each impasse even when the impasses occur in the same problem. In short, there seems to be some flexibility in whether patches are ignored, and perhaps also in how long they are retained.

Inter-test bug migration exhibits a more extensive use of patches. Inter-test bug migration is detected by testing students twice a short time apart (say, two days) with no intervening instruction. The student has a consistent bug on each test, but not the same bug. The bugs are related in that they can be generated by different repairs to the same impasse. It appears that the student has retained the procedure between the two tests, but the patch that was used on the first test was not retained. Instead, a new repair was selected, stored in a patch, and used consistently throughout the second test.

Bugs do not always migrate. Some bugs are held for months or years. Apparently, patches can be stored for long periods of time.

Bug migration was predicted in advance of its observation (Brown & VanLehn, 1980). It falls out as a natural consequence of viewing the repair process as modifying the execution (short term) state of the processor that interprets the stored procedure. The dynamic LPS hypothesis naturally predicts bug migration. What it has trouble with is explaining the repetition of the same repair to the same impasse, a phenomena referred to above as creating, storing and reusing a patch. The dynamic LPS hypothesis could explain this as a chance selection of the same repair over and over again. However, it is much more plausible to add an ancillary hypothesis that some kind of patch creation and storage exists. The patch hypothesis is difficult to verify since there is no way to tell whether or not a student has a patch (they could just have chosen the same repair twice). The only argument for their existence is intuitive plausibility. Nonetheless, encountering seventh graders with bugs that are acquired in the third grade is, for me, a fairly compelling demonstration that patches exist, even if they aren't a proper part of the model.

The static LPS hypothesis' account for bug migration

So far, bug migration has been discussed in terms of the dynamic LPS hypothesis. However, the static LPS hypothesis can generate the same predictions. The forté of static LPS is making permanent changes in the procedure. It examines the procedures's structure in order to find (or predict) impasses and install patches. This is all done without executing the procedure. The installation of a patch into the procedure naturally predicts stable, long-term bugs. Bug migration is more problematic. To predict bug migration, one must assume that it is possible to have stochastic patches: a random variable governs which action the patch will take. The various bugs in a bug migration class result from the patch switching at random among various sub-patches built by the local problem solver. This is somewhat implausible, perhaps.

However, the bare fact is that bug migration and long-term bugs both exists, and that dynamic and static both predict one naturally, but require a supplementary hypothesis to account for the other. The dynamic LPS hypothesis requires patch abstraction and storage; the static LPS hypothesis requires stochastic patches. So there is really no decisive argument here. We must look a little deeper.

Impasses and repairs need dynamic information

The preceding argument tried to relate the model's chronology, the sequence of derivational events, to real time. Such performance arguments are often quite slippery. Memory can always be used to shuttle hypothetical cognitive events forwards in time in order to satisfy the exigencies of the observations. Indeed, the argument above ended inconclusively. Laying time aside, the main difference between static LPS and dynamic LPS hypotheses is the kind of information available to the local problem solver. A dynamic local problem solver has the current state (i.e., active goals, a partially worked exercise). The static local problem solver has the procedure's calling structure (goal-subgoal hierarchy). The static local problem solver can perhaps examine all the failure modes of a primitive operator, such as decrement, and decide what to do for each one. However, there are intricate ways that a procedure can fail during execution. For the static local problem solver to find them, it would have to simulate running the procedure, and hence it would become, in effect, a dynamic local problem solver.

As an example, consider the bug Borrow-Across-Zero. Under the static LPS hypothesis, this bug is generated by assuming that the student has never learned how to borrow from zero, and that the static LPS has built into the procedure a patch so that when decrement fails by trying to decrement a zero, the focus of attention is shifted left to a nearby, non-zero digit, which is decremented instead of the zero. Figure 6–6a shows Borrow-Across-Zero solving a problem. Notice that when it borrows in order to answer the tens column, it must decrement the hundreds column a second time (problem state *e*). This creates a rather unusual combination of scratch marks. The very first time this student could have seen such scratch marks is the first time the student solved a BFZ problem. Until the student actually tackles the first BFZ problem, static local problem solving would have no reason to suspect such a strange double-decrement situation might arise.

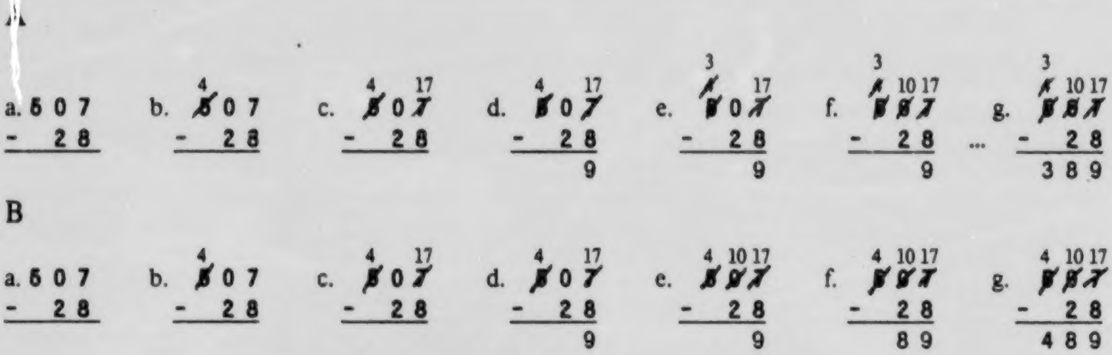


Figure 6–6
(A) Problem state sequence for Borrow-Across-Zero.
(B) Problem state sequence for the bug set (Borrow-Across-Zero !Touched-Zero-As-Ten).

Some students have a variant of the bug which indicates that, for them, double decrementing is an unusual enough action that it warrants repair. An attempt to decrement an already decremented digit causes an impasse. Figure 6–6b shows the bug set (Borrow-Across-Zero !Touched-Zero-Is-Ten). Just after problem state *d*, it attempts to decrement the hundreds column a second time. However, the student did not do the decrement, taking an impasse instead. The Noop repair was applied, causing the decrement to be skipped. The student then did the second part of borrowing, the addition of ten to the tens column (state *e*), and finished the problem correctly. This shows that double-decrementing can cause local problem solving. If the static LPS hypothesis is to account for this, it must assume that the LPS is very smart. The LPS has to plan ahead to realize that decrements might stack up in some unusual situations, and prepare a repair for this case. This is entirely implausible.

The preceding example showed that impass detection required dynamic (runtime) information. There is a similar argument that shows that repairs also need dynamic information. The argument involves the Backup repair. It can be shown (see section 1 of appendix 9) that Backup is most simply formulated as a modification of the execution state rather than a modification to the structure of the procedure. The argument rests on the fact that in certain situations where Backup has been observed, there are two instantiations of a certain goal, and Backup only goes to one of them. Static Backup can't discriminate among several dynamic instantiations of the same goal, but dynamic Backup can. Although a complex patch could be constructed by static local problem solving, it would essentially have to do exactly what a dynamic Backup would do anyway. So a dynamic version of Backup is the simplest, most natural way to handle these special cases of local problem solving.

Summary

The information that is available statically is just not sufficient to explain the kinds of local problem solving that occur. Local problem solving makes essential use of information that is naturally available at runtime. To generate the information statically would require such powerful simulation capabilities of the static local problem solver it would be come essentially equivalent to a dynamic local problem solver. So the static LPS approach is just not workable.

6.5 Formal hypotheses

The main ideas of local problem solving are impasses, repairs, their independence, and their embedding in the dynamic, runtime environment. To express these formally, a description of how procedures are executed is needed. That is, the gross architecture of a procedural interpreter will be used to formalize the theory. Chapter 3 began the formalization by postulating an undefined function, `Cycle`, that maps a runtime state into the next runtime state. It is used to describe the observable actions of the student when the student applies a procedure to a runtime state. Chaining applications of `Cycle` generates the procedure's solution to an exercise problem. In order to formalize local problem solving, `Cycle` will be defined in terms of several new, undefined functions. The nomenclature that will be used is:

`P`	A variable designating the student's procedure.
`S`	A variable designating the current runtime state.
`(Internal S)`	A function that returns the internal (execution, or interpreter) runtime state S.
`(External S)`	A function that returns the external (problem) runtime state of S. The runtime state is a composite of the internal and external state.
`(Cycle P S)`	A function that inputs a procedure and a runtime state and outputs a set of next states. It represents one cycle in the interpretation/execution of the procedure.
`(Interpret P S)`	An undefined function that expresses what the procedure does without local problem solving. It represents the "normal" interpretation of the procedure. It inputs a procedure and a runtime state and outputs the next runtime state. It represents one cycle of the interpreter. It will be defined later by the procedural representation language.
`(Repair S)`	An undefined function that takes a runtime state and returns a set of runtime states corresponding to various repairs.
`(Impasse S)`	An undefined predicate on states. It is true of a runtime state if the combination of execution and problem state constitutes an impasse.

The basic technique used to formalize local problem solving is the same as the one used to formalize learning. In this case, two undefined functions are used: `Repair` and `Impasse`. The predictions of the theory will be made in terms of them. Various constraints (hypotheses) will be placed upon them. The actual functions used in Sierra are just one way of instantiating the two functions in obedience to the constraints. In particular, `Repair` is formalized by a set of five repairs: Noop, Backup, Refocus, Force and Quit. The formalization of `Impasse` is similar. A set of *impasse conditions* is defined. For instance, precondition violations are one kind of impasse condition. Impasse conditions are to `Impasse` as repairs are to `Repair`.

Given the nomenclature, the basic architecture of the combined interpreter and local problem solver is defined by the following hypothesis:

Local problem solving

Let

```
(Cycle P S) ≡
if ~(Impasse S) then {(Interpret P S)}
   else {(Interpret P S') | S' ∈ (Repair S) and ~(Impasse S') }.
```

This hypothesis defines the `Cycle` function in terms of the three undefined functions, `Repair`, `Impasse`, and `Interpret`. `Interpret` is the normal interpretation of the procedure. If the current state is not an impasse, then `Interpret` is what `Cycle` does. If there is an impasse, then a repair is inserted before the `Interpret`. Because more than one repair is possible, there may be more than one successor state. Hence, `Cycle` returns a set of states.

There are several tacit features that are built into the definition of `Cycle`. Although it is redundant, it is useful to break these out as separate hypotheses. This makes it easier to refer to the concepts later.

> *Dynamic LPS*
> The local problem solver reads and changes the dynamic (execution time) state, but it does not change the procedure's structure.
>
> *Repair-impasse independence*
> Any repair can be applied to any impasse.
>
> *Filter-trigger symmetry*
> An impasse condition triggers local problem solving if and only if it also acts as a filter on repairs.

The first two hypotheses have already been discussed. The last one reflects the idea that repairs actually fix impasses. That is, `Impasse` must be true after the `Repair` function is done. It turns out that some repairs change the state in such a way that the new state is an impasse, perhaps of a different kind than before. That is, the repair doesn't really fix the problem; the interpreter is still stuck. Such repairs are filtered. To put it differently, if a certain impasse condition is sufficient to cause repairs (trigger local problem solving), then it is also effective in filtering repairs. All this follows from the basic notion that local problem solving really is a form of problem solving.

Chapter 7
Deletion

Two sources of bugs have been identified so far. One is overgeneralization, or rather correct induction from impoverished sets of examples (see section 6.1). The other source uses local problem solving to repair impasses, which are caused ultimately by incomplete learning (see section 6.3). In a sense, these two explanations fall under the broad headings of *learning* and *invention*. This chapter shows that a third source of bugs exists, something akin to *forgetting*. It perturbs the structure of a learned (core) procedure. For historical reasons, the new source of bugs is called deletion. This chapter discusses the reasons why the theory needs deletion. It presents a certain group of bugs and discusses three explanations for them:

1. The bugs result from local problem solving applied to procedures generated by partially completed learning.
2. The bugs result from of overgeneralization.
3. The bugs result from deletion of part of a learned procedure.

It is shown that neither of the first two explanations for the bugs work. This justifies introducing a new formal mechanism, deletion, into the theory.

7.1 Local problem solving will not generate certain bugs

Many bugs can be accounted for by incomplete learning followed by local problem solving. The basic idea is that the student is tested on skills that either have not been taught yet, or haven't been mastered. This often leads to impasses and repairs and, in turn, to bugs. However, this account will not work for certain bugs. For handy reference, these bugs will be called the deletion bugs. Explaining them is the target of this chapter.

The deletion bugs are best understood in contrast to bugs generated by local problem solving. The following bug can be generated by incomplete learning and repair:

Stops-Borrow-At-Zero:

$$\begin{array}{r} 3\;4\;6 \\ -\;1\;0\;2 \\ \hline 2\;4\;3 \end{array}\checkmark \qquad \begin{array}{r} 3\;\overset{3}{4}{}^{1}6 \\ -\;1\;2\;9 \\ \hline 2\;1\;6 \end{array}\checkmark \qquad \begin{array}{r} \overset{2}{3}{}^{1}0^{1}7 \\ -\;1\;6\;9 \\ \hline 1\;4\;8 \end{array}\times \qquad \begin{array}{r} 3\;0^{1}7 \\ -\;\;\;\;\;9 \\ \hline 3\;0\;8 \end{array}\times$$

The procedure behind this bug does not know how to borrow across zeros. It borrows correctly from non-zero digits, as shown in the second problem. On the third problem, it attempts to decrement the zero, hits an impasse, and repairs by skipping the decrement operation entirely (the Noop repair). The point is that this bug has a complete, flawless knowledge of borrowing from non-zero digits, but it doesn't know anything about borrowing from zero. Precisely at one of the lesson boundaries in the subtraction curriculum, its understanding stops. Now compare this knowledge state with the one implicated by the following bug:

Don't-Decrement-Zero:

$$\begin{array}{r} 3\,4\,5 \\ -\,1\,0\,2 \\ \hline 2\,4\,3 \end{array} \checkmark \qquad \begin{array}{r} 3\,\overset{3}{4}\,{}^{1}5 \\ -\,1\,2\,9 \\ \hline 2\,1\,6 \end{array} \checkmark \qquad \begin{array}{r} \overset{2}{3}\,{}^{1}0\,{}^{1}7 \\ -\,1\,6\,9 \\ \hline 1\,4\,8 \end{array} \times \qquad \begin{array}{r} \overset{2}{3}\,{}^{1}0\,{}^{1}7 \\ -\quad\ 9 \\ \hline 2\,10\,8 \end{array} \times$$

This bug also misses just BFZ problems ("BFZ" abbreviates "borrow from zero"). Indeed, it gets the same answer on the third problem as the previous bug, Stops-Borrow-At-Zero. However, it solves BFZ problems in a very different way. Notice the fourth problem. The borrow in the units column caused some, but not all, of the BFZ subprocedure to be executed. The following problem state sequence shows the initial problem solving:

$$\text{a.}\ \begin{array}{r} \overset{2}{3}\,0\,7 \\ -\quad\ 9 \\ \hline \end{array} \qquad \text{b.}\ \begin{array}{r} \overset{2}{3}\,{}^{1}0\,7 \\ -\quad\ 9 \\ \hline \end{array} \qquad \text{c.}\ \begin{array}{r} \overset{2}{3}\,{}^{1}0\,{}^{1}7 \\ -\quad\ 9 \\ \hline \end{array} \qquad \text{d.}\ \begin{array}{r} \overset{2}{3}\,{}^{1}0\,{}^{1}7 \\ -\quad\ 9 \\ \hline 8 \end{array}$$

Most of the BFZ subprocedure is there. What is missing is its last action, decrementing the ten in the ten's column to nine, which should occur between states *b* and *c*. Because the bug does some of the BFZ subprocedure, it is likely that subjects with this bug have been taught borrowing across zero. But it is also clear that they did not acquire all of the subprocedure, or else forgot part of it. If the subtraction curriculum was constructed so that teachers first taught one half of borrowing across zero and some weeks later taught the other half, then one would be tempted to account for this bug with incomplete learning. But BFZ is, in fact, always taught as a whole. So some other formal technique is implicated in this bug's generation. Don't-Decrement-Zero is one of the deletion bugs. Several others are detailed in chapter 10.

The case has been made that incomplete traversal of the curriculum will not generate a procedure that is appropriate for explaining this bug. Another way to make the same point is to note that repair could, in principle, generate the bug by using a Noop or Backup repair that would cause the tens-column decrement to be skipped. However, in order to have a repair, one must have an impasse. In this case, the impasse needs to be just before the decrement (i.e., between states *b* and *c* above). However, there is no apparent reason for an impasse there. The decrement is merely subtracting one from ten — an easy, unproblematic operation. No impasse condition that I know of will cause an impasse there. Without an impasse, there is no way to use Noop or other repairs to generate the bug. Again, the conclusion is that some other mechanism must be utilized to derive this bug.

7.2 Overgeneralization should not generate the deletion bugs

Overgeneralization can generate Don't-Decrement-Zero, but only at the cost of losing explanatory adequacy. The trick to an overgeneralization-based derivation is to induce that the decrement action in question is *optional.* One assumes that the student has received examples teaching BFZ. The examples will, of course, have the decrement action. For some reason, the student induces that this decrement is optional, even though all the examples happened to have it. On a test, the student instantiates the generalized subprocedure by choosing not to make the optional decrement. This generates the bug. The other deletion bugs can be generated with similar optionality-based inductions. Optionality is a plausible primitive concept for a procedural representation language to have, so there is nothing wrong a priori with this explanation.

The problem is with the nature of optionality. Inducing an optional fragment of a subprocedure is inducing a disjunction. The procedure acquires a choice about whether or not to execute the action. Thus, inducing a disjunction inside the subprocedure violates the one-disjunct-per-lesson hypothesis. To put it differently, it is a felicity condition that the teacher will show the student when a disjunction is needed. But all the examples used a decrement; none omitted it. The student has no evidence that a disjunction is needed. It is a direct violation of the felicity condition to put one in. To admit this violation just to generate a few bugs wrecks an otherwise explanatory framework.

7.3 The problems of defining a deletion operator

It has been shown that the two bug-generating pathways that the theory currently provides are ineffective in generating the deletion bugs. This motivates including a new operator in the theory. On the basis of the bug Don't-Decrement-Zero, it seems that some kind of deletion operator will do the job, something that removes an action such as decrement from a sequence of actions in a subprocedure.

It is not easy to formalize deletion in an empirically adequate way. Richard Young and Tim O'Shea used a model based on deleting production rules to generate some of the most common bugs (Young & O'Shea, 1981), including most of the deletion bugs. However, their approach could also generate star bugs. Given their production system, which has 22 rules, there are 22 possible rule deletions. However, only 7 of the possible 22 rule deletions generate bugs. Deleting certain of the other rules generates star bugs. In general, totally free, unconstrained deletion overgenerates wildly. One fix is to allow the theorist to specify which rules may or may not be deleted. This just transforms the question of why do only certain bugs exist, into the question of why do certain rules get deleted and not others. It doesn't explain very much. So the real problem with deletion is to put just the right constraints on it so that no star bugs are predicted and yet the deletion bugs are generated. Chapter 10 gives this tricky issue a full discussion.

For now, deletion will be formalized using an undefined function, `Delete`, which mutates procedures. It is interposed between the output of `Learn` and the input to `Cycle`. To capture this formally, the following hypothesis is used:

> *Deletion*
> If `P` is a core procedure, then all `P'` ∈ `(Delete P)` are core procedures as well.

The function `Delete` is set-valued to capture the fact that there is often more than one possible deletion that can be made to a procedure. As with the other main undefined functions in the theory, `Delete` will be defined by acquiring more constraints upon its behavior.

Chapter 8
Summary: Architecture Level

The preceding five chapters laid out the general architecture of the model and defended the main principles of the theory. This chapter summarizes both, makes a few comments, and introduces some of the issues discussed in following chapters.

8.1 The architecture of the model

The expositional strategy of this document is to start with an architecture composed of undefined functions, then to add constraints that gradually define the functions. At the top level, the architecture consists of three undefined functions, `Learn`, `Delete` and `Cycle`. `Learn` takes a lesson and a procedure as inputs; it returns a set of procedures. `Learn` represents the various ways that its input procedure can be augmented in order to assimilate the lesson. `Delete` takes a procedure as input; it returns a set of procedures as output, where each procedure is the result of deleting some part of the input procedure. In a later chapter, it will be shown that it simply deletes a rule from the And-Or graph that represents the procedure. The need of a deletion operator that is distinct from learning is argued for in chapter 7. The third undefined function, `Cycle`, represents one cycle in the interpretation/execution of a given procedure. Its inputs are a procedure and a "runtime state." A runtime state is a composite whose parts are an external state (i.e., a problem state) and an internal state (i.e., the interpreter's state). A runtime state represents the kind of information that can change while the procedure is running. The access function `(External S)` returns the external state of a given runtime state, S. Similarly, `(Internal S)` returns the interpreter's state of the S. The function `Cycle` computes the "next" runtime state. It takes a procedure and a runtime state, and it returns a set of runtime states. It returns a set because interpretation of the procedure is sometimes non-deterministic. Several states are possible "next" states. Given these functions, the top level of the model is defined by the following hypotheses:

> *Incremental Learning*
> Given a lesson sequence $L_1 ... L_n$ and an initial procedure P_0:
> Procedure P_i is a core procedure if
> (1) $P_i = P_0$, or
> (2) $P_i \in$ `(Learn` L_i P_{i-1}`)` and P_{i-1} is core procedure.
>
> *Deletion*
> If P is a core procedure, then all P′ ∈ `(Delete P)` are core procedures as well.
>
> *Predictions*
> If S_0 is the initial state such that `(External` S_0`)` is a test exercise, then the set of predicted problem state sequence for students with core procedure P is exactly the set
> $\{\langle(\texttt{External } S_0)...(\texttt{External } S_n)\rangle \mid \forall i\ S_i \in (\texttt{Cycle P } S_{i-1})\}$.

These hypotheses say that the basic architecture has two simple cycles. One cycle acquires a procedure, and the other cycle executes a procedure to solve a problem. The acquisitional cycle is called the *learner* in Sierra. It runs once for each lesson: First it executes `Learn` on a lesson, producing a set of procedures, then it executes `Delete`, which augments the set. The resulting set of core procedures is fed back as input procedures for the next cycle of the learner. The other cycle is called the *solver* in Sierra. It executes a core procedure to solve some problems. It includes both normal execution and local problem solving. Roughly speaking, the solver-cycle happens once for each action of the procedure. The grain of the cycle cannot be stated more precisely until the procedure representation language becomes more defined. The relationship between the learner and the solver are discussed further in section 2.1.

Observable predictions consist of problem state sequences. Each predicted student behavior is a set of problem state sequences, one sequence per test problem. (In particular, the solver generates intra-test bug migrations as well as stable bugs.) These whole-test sequences could be compared directly to student behavior. However, given the numbers of students, core procedures, and repairs involved, such a direct comparison would be an awesome task. A much simpler test of the theory is used. It is described in section 2.1.

8.2 Hypotheses and their support

The preceding formalisms serve basically as a framework on which more substantive hypotheses are hung. As mentioned, the expository tactic is to begin with undefined functions and slowly define them. Most of the preceding chapters was concerned with defining `Learn` and `Cycle`. This section presents each of the remaining "substantive" hypotheses.

> *Induction*
> If $P_i \in (\texttt{Learn}\ L_i\ P_{i-1})$ then for each example problem x in $(\texttt{Examples}\ L_i)$, the problem state sequence that is P_i's solution to x is equal to the problem state sequence that is the solution to x used in the example.

This hypothesis says that mathematical skill acquisition is inductive in character. The arguments supporting it are in chapter 3. Inductive learning is the only learning framework of those discussed that is consistent with the gross features of classroom learning. The induction hypothesis is framed carefully to work with the deletion hypothesis, which was stated above. Although both `Learn` and `Delete` produce core procedures, the core procedures that `Delete` produces do not generalize the lessons' examples. In a rough sense, the sequence of actions produced by a deleted procedure is the same as the sequence of actions from the undeleted procedure except that a few of the actions are removed. `Delete`-produced core procedures are not consistent with the lessons, but `Learn`-produced ones are.

> *Show-work*
> In worked examples of a lesson, all objects mentioned by the new subprocedure are visible, unless the lesson is marked as an optimization lesson.

This hypothesis expresses a key felicity condition. Students act as if they believe that the teacher will always "show all the work" while doing example exercises. For the portion of the procedure that introduces the new subprocedure, the teacher is expected to write down intermediate results that are normally held mentally. For instance, the first lesson on carrying has examples like this one:

$$\begin{array}{cc|c} & \underline{\text{tens}} & \underline{\text{units}} \\ & 3 & 7 \\ + & & 5 \\ \hline & 3 & 12 \end{array}$$

$$\underline{42}$$

The units column sum, 12, appears explicitly rather than being held mentally as it will be when carrying is eventually mastered. Such mastery is taught in separate lessons that show how to avoid some of the writing by holding intermediate results mentally. These lesson are specially marked. The theory's term for them is *optimization* lessons. Sierra doesn't handle optimization lessons, mostly because optimization lessons are not used in teaching subtraction and the other skills that data are available on. The show-work hypothesis is one solution to the invisible objects problem of inductive learning (see chapter 5). Other solutions have the same empirical adequacy as show-work, but they have less explanatory adequacy. They do not explain why teachers almost always show their work, nor do they explain why there are two kinds of lessons.

In order to formalize the next felicity condition, it is convenient to use three new undefined functions. The formerly undefined function `Learn` will be defined in terms of them. The three new functions are listed below, with informal explanations. Two simple helping functions are defined as well.

`(Induce P XS)`	represents disjunction-free induction. The first argument, `P`, is a procedure. The second, `XS`, is a set of worked example exercises. The function returns a set of procedures. Each procedure is a generalization of `P` that will solve all the exercises the same way that they are solved in the examples. `Induce` is not permitted to introduce disjunctions. If the procedure cannot be generalized to cover the examples, perhaps because a disjunction is needed, then `Induce` returns the null set.
`(Disjoin P XS)`	represents the introduction of a disjunction (e.g., conditional branch) into `P`, the procedure that is its first argument. The second argument, `XS`, is a set of examples. `Disjoin` returns a set of procedures. Each procedure has had one disjunction introduced into it. The disjunction is chosen in such a way that `Induce` can generalize the procedure to cover all the examples in `XS`. If there is no way to introduce a single disjunct that will allow all the examples to be covered, then `Disjoin` returns the null set.
`(Practice P XS)`	represents another kind of disjunction-free generalization, one driven by solving a set of practice exercises, `XS`. `Practice` returns a set of procedures, each one a generalization of its input procedure `P`.
`(Examples L)`	This access function returns the sequence of worked examples of the lesson L.
`(Exercises L)`	This access function returns the sequence of practice exercises of the lesson L.

Given these functions, the remaining felicity condition can be simply stated:

One-disjunct-per-lesson

Let

```
(Learn P L) ≡
  If (Induce P (Examples L))≠{} then (Learn1 P L)
     else (Learn2 P L).
```

where

```
(Learn1 P L) ≡
  { P" | ∃ P' such that P' ∈ (Induce P (Examples L))
              and P" ∈ (Practice P' (Exercises L)) }.
```

and

```
(Learn2 P L) ≡
  { P" | ∃ P' such that P' ∈ (Disjoin P (Examples L))
              and P" ∈ (Learn1 P' L) }.
```

Moreover, `(Induce P XS)` and `(Practice P XS)` do not introduce into P any new disjunctions or any new disjuncts on old disjunctions, and `(Disjoin P XS)` inserts into P exactly one new disjunction or one new disjunct on an old disjunction.

The first part of the hypothesis says, essentially, that `Learn` performs the functions `Disjoin`, `Induce` and `Practice`, in that order. However, `Disjoin` is skipped if it is unnecessary for the particular lesson. The last two clauses of the hypothesis express a key idea: Students learn at most one subprocedure (disjunct) per lesson. Put differently, the students act as if they believe that the teacher has designed the lesson sequence in such a way that introduction of a new disjunct (subprocedures) always falls on a lesson boundary. Some subprocedures (disjuncts) may take several lessons to learn, but no lesson introduces more than one. The arguments for the hypothesis is in chapter 4.

This felicity condition is one solution to the disjunction problem of inductive learning. Although one of the competing hypotheses to it is just as empirically adequate as it, one-disjunct-per-lesson has the added value that it explains why lessons are so often used and why they are helpful when they are used. The other approach would work equally well with a homogeneous sequence of examples rather than the partitioned sequence, defined by the lesson boundaries, that is actually used. Since the other approach ignores lessons, it can't explain why the lesson convention has been universally adopted as a helpful educational framework. Since the felicity condition does explain the use of lessons, it has greater explanatory adequacy.

In order to state the remaining hypotheses, three more undefined functions will be introduced. They will be used to define `Cycle`, thereby allowing the architecture of the local problem solver to be cleanly expressed.

`(Interpret P S)`	represents one cycle of the normal interpretation (execution) of the procedure P. The second argument, S, is a runtime state. `Interpret` returns the next runtime state. `Interpret` is defined by the representation language used for procedures.
`(Impasse S)`	is a predicate that is true when the runtime state S is an impasse. It is considered to be implemented by a set of *impasse conditions.* If any impasse condition is true, then `Impasse` is true. `Impasse` represents the problem detection component of local problem solving.
`(Repair S)`	represents the other half of local problem solving, repair. It is considered to be implemented by a set of *repairs,* such as Noop and Backup. `Repair` returns a set of runtime states. Each state results from the action of one of the repairs on the input state S.

With these functions, it is simple to state the main hypothesis governing local problem solving.

Local problem solving

```
Let
(Cycle P S) ≡
if ~(Impasse S) then {(Interpret P S)}
  else {(Interpret P S') | S' ∈ (Repair S) and ~(Impasse S')}
```

The usual cycle is simply to execute the function `Interpret` once. However, if `Impasse` is true, then an execution of `Repair` is inserted before the `Interpret`. `Repair` outputs a set of states. Some of these are filtered: if `Impasse` is true of a state, then that state is not passed to `Interpret`. Usually, several of `Repair`'s output states are left after filtering. Hence, the execution cycle becomes non-deterministic at this point. Several ideas behind this hypothesis are so important that it is best to break them out separately, as "corollaries" of the main hypothesis, so that they can be easily referenced later.

Repair-impasse independence
Any repair can be applied to any impasse.

Filter-trigger symmetry
An impasse condition triggers local problem solving if and only if it also acts as a filter on repairs.

These two corollaries emphasize that local problem solving really is problem solving, where the problem is being stuck. The problem is not solved until the procedure is unstuck (filter-trigger symmetry). Moreover, it doesn't matter how one gets unstuck as long as one succeeds (repair-impasse independence). The arguments for local problem solving are presented in chapter 6. It is shown that the theory could do without it and still generate some bugs, but in doing so it would lose much of its ability to explain those bugs. Essentially, it would have to build certain observed bugs explicitly into the set of primitives that are assumed to be present before learning begins. Thus, it offers no explanation for why these bugs occur and not others. Another "corollary" of the local problem solving hypothesis is

Dynamic LPS
The local problem solver reads and changes the dynamic (runtime) state, but it does not change the procedure's structure.

This says that impasses and repairs effect the runtime state rather than the procedure's structure. The kind of information that impasses and repairs need is available at runtime but not statically. The arguments for this hypothesis are in section 6.4.

There is another key feature of local problem solving that needs mentioning despite the fact that I have no defense to give for it. It is difficult to state formally, although the basic idea is clear. The repairs that have been discovered so far are extremely simple local changes to the interpreter's state. Also, impasse conditions are unsophisticated, local checks. The local problem solver does not seem to do any large computations, nor does it look ahead to see the consequences of its actions or the interpreter's actions. The local problem solver doesn't really go looking for trouble, but when it encounters some, it just barges through it expending as little work as possible. To summarize this general impression, it is convenient to name the hypothesis:

Locality
The repairs and impasse conditions are local.

This doesn't really constrain the model so much as express an orientation or direction in the ongoing endeavor of making local problem solving more precise.

8.3 Commentary on the arguments and inherent problems

The arguments presented in the preceding chapters have nothing to do with the way the principles were actually discovered. The tales of the principles' discoveries deserve to be told late at night over a couple of beers, if at all. The supporting arguments were constructed more recently. A main task of their construction was the discovery and understanding of the problems that the principles solve. This was sometimes a nontrivial task. For instance, it was plain to see what happened when the show-work hypothesis was turned off in Sierra: the model overgenerated wildly. But it was not clear whether this was a problem with the particular knowledge representation being used or whether the explosion was due to a more general problem. It appears now to be a general problem, labelled the invisible objects problem. It seems to be a problem that affects *any* inductive account of learning, despite the fact that it has slipped by unnoticed in virtually all AI work on inductive learning. For lack of a better word, such general problems will be labelled *inherent* problems, because they seem inherent in any study of the domain.

Three kinds of inherent problems occurred in the preceding chapters. One inherent problem was figuring out how much of the classroom experience could be ignored. There isn't much to say about this problem. One takes a broad look at the phenomena and their context, makes a guess, and constructs a theory. In this case, it is fairly clear that induction is a reasonable guess. For skills other than mathematics, it may be much less clear which frameworks will yield successful theories.

A second kind of inherent problem involves what one could loosely call *laws of information*. The problems seem to be inherent to any thinker, mechanical or human, that performs the given information processing task. In this case, two inherent problems with induction were encountered: the disjunction problem and the invisible objects problem. Such informational problems are extremely subtle. They are subtle in two ways. First, it is hard to discover that the problems are there and what their exact nature is. For instance, the disjunction problem, which is well known to philosophers, has not been generally acknowledged by AI researchers until recently. Some linguists still tend to misunderstand it as a problem concerning the presence or absence of negative examples (see section 4.1). Information is apparently very slippery stuff. One can get buried in the formalisms used to express and manipulate it, so buried that a whole learning machine can be constructed without ever realizing that one has somehow solved an informational problem or even that the problem was there at all.

The second subtlety with informational problems comes out clearly in the arguments of chapters 4 and 5. Certain solutions to the disjunction problem and the invisible objects problem are often extremely difficult to differentiate on empirical grounds. For instance, it is difficult to differentiate the hypothesis that learners introduce the minimal number of disjuncts from the hypothesis that they introduce at most one disjunct per lesson. To split these hypotheses is not possible with the current lesson sequences since the hypotheses make identical predictions using them. Empirical tests that could differentiate the hypotheses would require difficult and morally questionable educational experiments. The subtlety of splitting hypotheses about how people solve informational problems makes sense in the context of the collective experience with computer programming. It is an axiom of programming that there are many ways to solve an information processing problem. Some may perform very similarly despite significant underlying differences. By analogy, there must be many possible solutions to human information processing problems. It will not always be simple to tell which one(s) people use.

The third kind of inherent problem is relatively straightforward. There are certain patterns in the data that stick out like sore thumbs. The problem is to account for them. Three major patterns were discussed in the preceding chapters: overgeneralization bugs (section 6.1), Cartesian product bugs (section 6.3), and deletion bugs (section 7.1). Each has an intuitively compelling explanation. In these cases, the explanations are based on overgeneralization, invention and forgetting, respectively. However, closer examination reveals that the phenomena can be accounted for by other means. In fact, any one of the three mechanism — overgeneralization, invention, and forgetting — can account for all three patterns. However, in doing so, they take on an unconstrained, stretched aspect. Stretching the mechanisms to cover phenomena for which they are ill-suited leads to a lack of explanatory adequacy.

In short, solving even the simplest inherent problems in the theory requires appealing to explanatory adequacy. This was quite a surprise to me. Before the arguments were carefully worked out, I had expected empirical evidence to resolve most of the arguments. In fact, it does do most of the work, but it seems always to fall a little short of eliminating the last one or two competitors.

8.4 Preview of Part 2

Part 2, the representation level, consists of chapters 9 to 16. It tackles the problems of knowledge representation, although details of the syntax of the knowledge representation are delayed until the next level. The representational level addresses issues concerned with capabilities and expressive power. It addresses questions such as: should procedures be finite state automata, stack automata, or something more exotic? Should patterns have the full descriptive power of a first-order logic? How much flexibility should there be in storing and restoring focus of attention? What about "short-term memory" for numbers?

The organization of the exposition divides representational issues along fairly traditional computer science lines. The first two chapters discuss *control flow* and *data flow*. The terms are taken from Rich and Shrobe (1976), who adopted these concepts in order to analyze programs from a language-independent position. (A more common use of the term "data flow," as in data flow computer languages (Dennis, 1974), has different connotations than the ones intended here.) The tack taken in these two chapters is to find out what constraints should be placed on the representation's ability to express control flow and data flow, and indeed, whether they should be separated at all. Chapter 13 concerns how procedures should *interface* with the external world. The external world of a computer program is usually an operating system, and the interface to it is notoriously ad hoc. For the procedures of mathematical calculation, the external world is a writing surface, such as a piece of paper. The interface is concerned with how that resource is addressed, read and written. For instance, how is the paper searched to locate information fitting a partial specification? To answer this question, the interface chapter describes the patterns that can be used for specifications and the kind of searches that can take place.

The modularity hypothesis and argumentation

Representational questions such as the ones above are extremely general. They can be construed to cut across many task domains (e.g., the issue of working memory). One way to argue these issues is to refer to results from all over psychology. Thus, results from digit span experiments would be used to justify a particular choice of working memory, e.g., a buffer with 7±2 cells. This style of argumentation was pursued by Newell and Simon in their work on human problem solving (1972). I doubt that I could improve on that magnificent accomplishment.

However, the underlying premise of that style of argumentation is that it is valid to use results from all sorts of tasks to argue for a particular information processing task. In particular, the assumption is that the mind is like a general-purpose computer: the mind uses the same architecture to perform a huge variety of tasks by loading itself with different programs. This premise has recently come under heavy fire in cognitive science.

Fodor, Chomsky and others of the MIT school of cognitive science have argued that it makes just as much sense to assume that the mind is modular (Fodor, 1983). Their claim is that mental architectures are specialized for the processing that they do. By analogy with programs, the modularity hypothesis is plausible. Computer science has found that there are some things that the vonNeuman architecture (the one used by most computers) is poor at, such as certain kinds of pattern recognition. Yet other architectures have been devised that do such tasks rapidly with simple programs. If the modularity hypothesis is true, then the style of argumentation used by Newell and Simon is no longer valid. To be trustworthy, an argument can use data only from the task at hand. This is exactly what the arguments of the next level do.

Registers and stacks

One of the earliest and most fundamental changes in computer programming languages was the move from register-oriented languages to stack-oriented languages. In register-oriented languages, one represents programs as flow charts or their equivalent. The main structure for regulating control flow is the conditional branch. Data flow is implemented as changes to the contents of various registers. Stack-oriented languages added the idea of a subroutine: something that could be called from several places and when it was finished, control would return to the caller of the subroutine. While the register-oriented languages need only a single pointer to keep track of the control state of the program, stack-oriented languages need a last-in-first-out stack so that the interpreter can tell not only where control is now (the top of the stack), but where it is to return to when the current subroutine gets done (the next pointer on the stack), and so on. The shift of computer science to stack-orientation also augmented the representation of data flow. A new data flow facility was to place data on the stack, as temporary information associated with a particular invocation of a subroutine. In particular, subroutines could be called recursively with parameters (arguments).

The fundamental distinction between register-orientation and stack-orientation has lapsed into historical obscurity in computer science, but surprisingly, psychology seems to be somewhat slow in making the transition. When a psychologist represents a process, it is frequently a flow chart, a finite state machine or a Markov process that is employed. Even authors of production systems, who are often computer scientists as well as psychologists, sometimes give that knowledge representation a register orientation: working memory looks like a buffer, not a stack, and productions are often not grouped into subroutines. For some reason, when psychologists think of temporary memory, whether for control or data, they think of global resources, such a registers. This tradition shows signs of changing. More recent production system architectures, such as Anderson's ACTF (1982), use goal stacks, subroutines, argument passing, etc.

There are well known mathematical results concerning the relative power of finite state automata, register automata and push down automata. Some of these results have been applied to mental processes such as language comprehension (see Berwick (1982) for a review). However, I find myself rather unconvinced by such arguments. As Berwick and others have pointed out, these arguments must make many assumptions to get off the ground, and not all of them are explicitly mentioned, much less defended.

The project of these chapters is to argue for a modern representation of core procedures based on a rich structure of motivated assertions: the hypotheses of the architecture level. The discussions in the architecture level did not make strong assumptions about the representation language because they dealt with the facts at a medium-high level of detail. The arguments in the representational level show what must be assumed of the representation language in order to push the structure of the architecture down to a low enough level that precise predictions can be made, and made successfully. That is, they show what aspects of the knowledge representation are *crucial* to the theory.

Chapter 9
Control Flow

The objective of this chapter is to show that the control structure is recursive. The argument starts with a minimum of assumptions about control. Instead, the hypotheses on local problem solving and subprocedure acquisition from earlier chapters will be used. However, it is necessary to speak in an informal way of goals and subgoals, with the intention that these be taken as referring to the procedural knowledge of subtraction itself, rather than expressions in some particular representation (e.g., production systems, And-Or graphs, etc.). In particular, it will be assumed that borrowing is a subgoal of the goal of processing a column, and that borrowing has two subgoals, named borrowing-into and borrowing-from. Borrowing-into is performed by simply adding ten to a certain digit in the top row, while the borrowing-from subgoal is realized either by decrementing a certain digit, or by invoking yet another subgoal, borrowing-from-zero. These assumptions, or at least some assumptions, are necessary to begin the discussions. They are some of the mildest assumptions one can make and still have some ground to launch from.

Three control regimes are considered in this chapter:

1. *Finite state automata:* The internal, execution state for the core procedure is limited to a single "you are here" pointer. It indicates which state (or goal, or rule, or other construct in the procedural representation) is currently executing. The procedure may or may not be structured hierarchically. However, if it is, it may not have self-embedding subprocedures, i.e., subprocedures that call themselves recursively either directly or via other subprocedures.
2. *Push down automata:* The internal, execution state for the procedure contains a last-in, first-out *goal stack*. The stack stores the currently executing goal's state by *pushing*. It resets the control state to a saved goal by *popping*. The procedure's structure may have recursive subprocedures.
3. *Coroutines:* Coroutines are independent parallel processes, each roughly equivalent to a push down automaton. They are taken as a representative of the class of higher-order control structures.

The third alternative isn't considered as seriously as the others. The main competition is between finite state and push down automata.

Formal automata results

There are formal results concerning the expressive power of these control regimes. It can be proved that there are certain tasks that can be accomplished by procedures written for push down automata, and yet no procedure written for a finite state automata can perform the tasks in their full generality. These formal results are irrelevant here for several reasons: (1) The procedures for mathematical skills can be easily expressed for finite state automata. (2) The formal expressability arguments turn on the fact that a push down automaton's stack can be infinitely large. A push down automaton with a finite upper bound on its stack length is equivalent in power to a finite state automata. An infinite stack is physically impossible to implement on material information processors, including brains and digital computers. There are no true push down automata in the material world.

These trite observations show the impotency of expressability arguments for empirical theories (but not for mathematical ones). The psychologically interesting issues concern how closely the automata's architectures approximate the structure of the mind's information. To put it differently, the question is which control structure best fits the observed procedure, where "fit" is evaluated by seeing whether the control structure enables the procedure's representation to be simple while capturing the empirical evidence.

This chapter offers two kinds of arguments. One concerns local problem solving and the other concerns learning. Both arguments show that a stack-based architecture simplifies their respective components, the local problem solver and the learner, while capturing the empirical facts in a natural way. The arguments concerning the local problem solver are rather complex. Despite the fact that they are some of the strongest and most elegant arguments in the whole document, they are also the longest, so they have been moved to an appendix (appendix 10, sections 1 and 2). Only a synopsis will be presented here.

9.1 Chronological, Dependency-directed and Hierarchical backup

Control structure is not easily deduced by observing sequences of writing actions. Too much internal computation can go on invisibly between observed actions for one to draw strong inferences about control flow. What is needed is an event which can be assumed or proven to in some sense be the result of an elementary, indivisible control operation. The instances of this event in the data would shed light on the basic structures of control flow. Such a tool is found in a particular repair called the Backup repair. It bears this name since the intuition behind it is the same as the one behind a famous strategy in problem solving: backing up to the last point where a decision was made in order to try one of the other alternatives. This repair is crucial to the argument, so it is worth a moment to introduce it.

Figure 9-1 is an idealized protocol of a subject who has the bug Smaller-From-Larger-Instead-of-Borrow-From-Zero. The (idealized) subject does not know about borrowing from zero. When he tackles the problem 305–167, he begins by comparing the two digits in the units column. Since 5 is less than 7, he makes a decision to borrow (episode *a* in the figure), a decision that he will later come back to. He begins to tackle the first of borrowing's two subgoals, namely borrowing-from (episode *b*). At this point, he gets stuck since the digit to be borrowed from is a zero and he knows that it is impossible to subtract a one from a zero. He's reached an impasse. The Backup repair gets past the decrement-zero impasse by "backing up," in the problem solving sense, to the last decision which has some alternatives open. The backing up occurs in episode *c*, where the subject says, "So I'll go back to doing the units column." He takes one of the open alternatives, namely to process the units column in a normal, non-borrowing way. Doing so, he hits a second impasse, saying, "I still can't take 7 from 5," which he repairs ("so I'll take 5 from 7 instead"). He finishes up the rest of the problem without difficulty. His behavior is that of Smaller-From-Larger-Instead-of-Borrow-From-Zero.

a. $$\begin{array}{r} 305 \\ -\,167 \\ \hline \end{array}$$

In the units column, I can't take 7 from 5, so I'll have to borrow.

b. $$\begin{array}{r} 305 \\ -\,167 \\ \hline \end{array}$$

To borrow, I first have to decrement the next column's top digit. But I can't take 1 from 0!

c. $$\begin{array}{r} 305 \\ -\,167 \\ \hline 2 \end{array}$$

So I'll go back to doing the units column. I still can't take 7 from 5, so I'll take 5 from 7 instead.

d. $$\begin{array}{r} \overset{2}{\cancel{3}}05 \\ -\,167 \\ \hline 2 \end{array}$$

In the tens column, I can't take 6 from 0, so I'll have to borrow. I decrement 3 to 2 and add 10 to 0. That's no problem.

e. $$\begin{array}{r} \overset{2}{\cancel{3}}05 \\ -\,167 \\ \hline 142 \end{array}$$

Six from 10 is 4. That finishes the tens. The hundreds is easy, there's no need to borrow, and 1 from 2 is 1.

Figure 9–1
Pseudo-protocol of a student performing the bug
Smaller-From-Larger-Instead-of-Borrow-From-Zero.

From the pseudo-protocol, it is clear that the Backup repair sends control back to *some* previous decision point so that a different alternative can be pursued. The critical question is, what determines the decision point that Backup will return to? There are three well known backup regimes used in AI:

1. *Chronological Backup:* The decision that is returned to is the one made most recently, regardless of what part of the procedure made the decision.

2. *Dependency-directed Backup:* A special data structure is used to record which actions depend on which other actions. When it is necessary to back up, the dependencies are traced to find an action that doesn't depend on any other action (an "assumption" in the jargon of Dependency-directed backtracking). That decision is the one returned to.

3. *Hierarchical Backup:* To support Hierarchical Backup, the procedure representation language must be hierarchical in that it supports the notion of goals with subgoals, and the interpreter must employ a goal stack. In order to find a decision to return to, Backup searches the goal stack starting from the current goal, popping up from goal to supergoal. The first (lowest) goal that can "try a different method" is the one returned to. Such a goal must have subgoals that function as alternative ways of achieving the goal, and moreover, some of these alternative methods/subgoals must not have been tried by the current invocation of the goal. When Backup finds such a goal on the stack, it resets the interpreter's stack in such a way that when the interpreter resumes, it will call one of the goal's untried subgoals. (In AI, this is not usually thought of as a form of Backup. It is sometimes referred to by the Lisp primitives used to implement it, e.g., `THROW` in Maclisp, and `RETFROM` in Interlisp.)

The key difference among these backup regimes is, intuitively speaking, which decision points the interpreter "remembers." These establish which decision points the Backup repair can return to. In Chronological and Dependency-directed backtracking, the interpreter "remembers" all decision points. In Hierarchical backup, it forgets a decision point as soon as the corresponding goal is popped. The critical case to check is whether students ever back up to decision points whose corresponding goals would be popped if goal stacks were in use. If they don't return to such "popped" decision points, then Hierarchical Backup is the best model of their repair regime. On the other hand, if students do return to "popped" decisions, then either Chronological or Dependency-directed Backup is the better model. Evidence is presented in appendix 10 that students never back up to popped decision points. By "never," I mean that returning to popped decision points generates star bugs. This evidence vindicates Hierarchical backup, and shows that (1) procedures' static structure has a goal-subgoal hierarchy, and (2) a goal stack is used by the interpreter in executing the procedure. In short, push down automata are better models of control structure than finite state automata.

9.2 One-disjunct-per-lesson entails recursive control structure

There is a second argument for a recursive control regime. It shows that recursive control structure is necessary for the one-disjunct-per-lesson hypothesis to be true. That is, if the language can't use recursion, then the one-disjunct-per-lesson hypothesis cannot be imposed without causing the theory to lose empirical adequacy. The argument involves learning a certain way to borrow across zero, one that borrows center-recursively. In fact, it is the most widely taught BFZ (i.e., borrow from zero) method. It has been used throughout this document for examples. It is exemplified in the following problem state sequence:

$$\begin{array}{r} \\ 3\,{}^{1}0\ 5 \\ \underline{-\ 1\ 2\ 9} \\ \end{array} \qquad \begin{array}{r} 2 \\ 3\,{}^{1}0\ 5 \\ \underline{-\ 1\ 2\ 9} \\ \end{array} \qquad \begin{array}{r} 2\ 9 \\ 3\,{}^{1}0\ 5 \\ \underline{-\ 1\ 2\ 9} \\ \end{array} \qquad \begin{array}{r} 2\ 9 \\ 3\,{}^{1}0\,{}^{1}5 \\ \underline{-\ 1\ 2\ 9} \\ \end{array} \qquad \begin{array}{r} 2\ 9 \\ 3\,{}^{1}0\,{}^{1}5 \\ \underline{-\ 1\ 2\ 9} \\ 6 \end{array} \qquad \begin{array}{r} 2\ 9 \\ 3\,{}^{1}0\,{}^{1}5 \\ \underline{-\ 1\ 2\ 9} \\ 1\ 7\ 6 \end{array}$$

The zero has ten added to it, then the three is decremented, then the newly created ten is decremented.

The claim is that the only way to learn this way of borrowing in a non-recursive language violates the one-disjunct-per-lesson hypothesis. To make the argument concrete, a particular non-recursive language, namely flow charts, is used. Figure 9–2a shows borrowing from a core procedure that only knows how to borrow from non-zero digits. Figure 9–2c shows borrowing after borrowing across zero, in the fashion above, has been learned. Clearly, there are two branches to learn. One moves control leftward across a row of zeros, and the other moves back across them until the column originating the borrow is found (i.e., the Home? predicate is true of the column B). There are many other ways that borrowing could be implemented, but if recursive control is not available, they would all have to have two loops — one for traversing columns leftward, and one for traversing columns rightward.

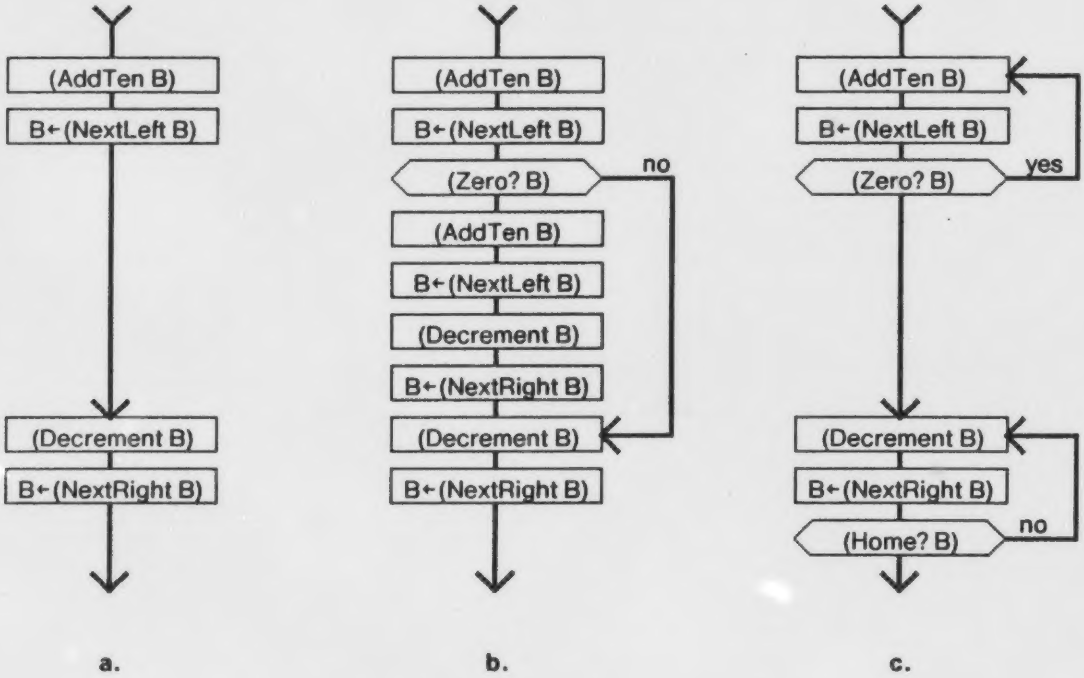

(AddTen B)
B←(NextLeft B)
(Decrement B)
B←(NextRight B)
a.
(AddTen B)
B←(NextLeft B)
(Zero? B)
no
(AddTen B)
B←(NextLeft B)
(Decrement B)
B←(NextRight B)
(Decrement B)
B←(NextRight B)
b.
(AddTen B)
B←(NextLeft B)
(Zero? B)
yes
(Decrement B)
B←(NextRight B)
(Home? B)
no
c.

Figure 9–2
A non-recursive representation of borrowing from zero.

Disjunction-free learning sanctions the acquisition of at most one branch, and this must be one that adjoins the newly learned subprocedure to the older material. A formal definition of branches and adjunction depends on the syntax of the language, but the essence of it should be clear by examining the difference between figures 9–2a and 9–2b. Functionally, the difference is that the core procedure of figure 9–2b has learned how to borrow from *one* zero. Syntactically, there is one branch, and it is an adjoining branch because one arm of the branch skirts the new material. The essence of adjunction is that one arm of the new conditional replicates the old procedure's control pathway.

It should be clear that the transition from 9–2a to 9–2c requires adding two disjunctions and thus violates the one-disjunct-per-lesson hypothesis. It is less clear, but equally true, that going from 9–2b to 9–2c requires adding two disjunctions (plus deleting some material as well). Two disjunctions must be introduced at exactly the lesson where the procedure goes from an ability to borrow across some finite number of zeros, to an ability to borrow across an arbitrary number of zeros. So, the finite state architecture forces the learner to violate the one-disjunct-per-lesson hypothesis.

Yet, if the language allowed recursion, then the BFZ goal could be represented as in figure 9–3b, with a recursive call to itself (the heavy box labelled "Borrow"). This representation allows the transition from non-zero borrowing (9–3a) to borrowing-from-zero (9–3b) to obey the one-disjunct-per-lesson hypothesis. In short, the language must have a recursive control structure so that a certain acquisitional transition obeys the one-disjunct-per-lesson hypothesis.

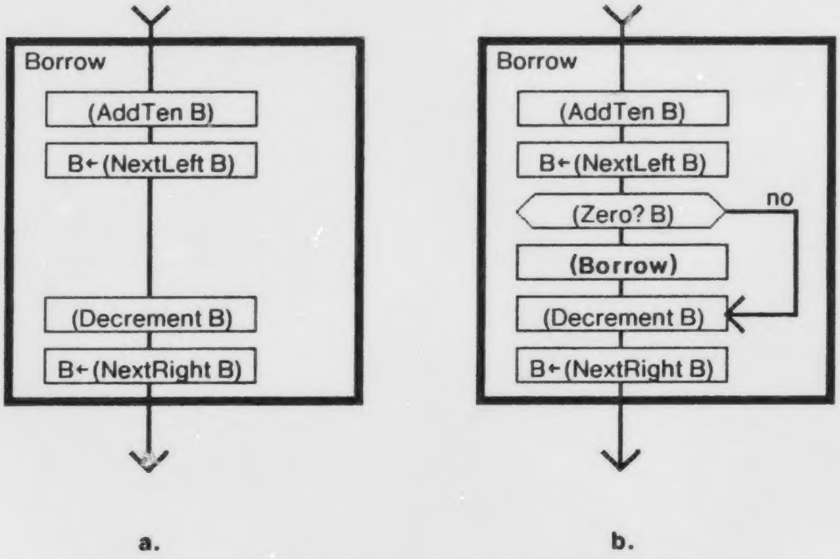


Figure 9–3
A recursive representation of borrowing from zero.

9.3 More powerful control structures

There are more powerful control regimes than stack-based ones. However, they introduce extra flexibility into the way control can be expressed. This extra flexibility is not only unnecessary, but it can cause the theory to make spurious, absurd predictions. As an example of the trouble more powerful control regimes cause, consider a simple control regime: Coroutines are a control structure that allows independent processes, each with their own stack. This control structure increases the expressiveness of the language, which allows acquisition to generate absurd core procedures. To demonstrate this point, consider the learning of simple borrowing. The instructional sequence has a problem state sequence such as:

$$\begin{array}{rrrr}
 & {}^{3}\phantom{{}^{1}5} & {}^{3}\phantom{{}^{1}5} & {}^{3}\phantom{{}^{1}5} \\
4{}^{1}5 & 4{}^{1}5 & 4{}^{1}5 & 4{}^{1}5 \\
\underline{-\,2\,9} & \underline{-\,2\,9} & \underline{-\,2\,9} & \underline{-\,2\,9} \\
 & & 6 & 1\,6
\end{array}$$

Given that coroutines are allowed, one way to construe the first two actions, the new ones, is that they are a new coroutine. It happens that the example has this coroutine executing before the old one, but the learner need not take that as necessary. The core procedure could execute the coroutines interleaved, as in

$$\begin{array}{rrrr}
 & & {}^{3}\phantom{{}^{1}5} & {}^{3}\phantom{{}^{1}5} \\
4{}^{1}5 & 4{}^{1}5 & 4{}^{1}5 & 4{}^{1}5 \\
\underline{-\,2\,9} & \underline{-\,2\,9} & \underline{-\,2\,9} & \underline{-\,2\,9} \\
 & 6 & 6 & 1\,6
\end{array}$$

That is, the first action of the new coroutine occurs, then the first action of the old coroutine. Next, the new coroutine resumes, and the borrow-from action occurs. Lastly, the old coroutine resumes, and answers the tens column. There are other ways that the interleaving could happen. If the old coroutine finishes before the new one, then the problem is answered incorrectly, because the borrow-from happens after the answer is written down:

$$\begin{array}{rrrr}
 & & & {}^{3}\phantom{{}^{1}5} \\
4{}^{1}5 & 4{}^{1}5 & 4{}^{1}5 & 4{}^{1}5 \\
\underline{-\,2\,9} & \underline{-\,2\,9} & \underline{-\,2\,9} & \underline{-\,2\,9} \\
 & 6 & 2\,6 & 2\,6
\end{array}$$

These are all ways of executing the same core procedure, one that was learned from an entirely correct sequence of actions. This core procedure seems an absurd prediction to make, a star bug. In short, the use of a more powerful control regime allows acquisition to generate core procedures that it should not. Restricting the control regime to be a stack improves the empirical adequacy of the theory by blocking the star bug.

9.4 Summary and formalization

Two arguments have been presented showing that a goal stack is necessarily a part of the execution state and that procedures should employ a goal hierarchy. A third argument made the case that it was inadvisable to have a more flexible control structure than a simple stack-based one. The conclusion is that procedures should be represented with a recursive control structure and that the interpreter should use a goal stack.

There are many formal languages for representing procedures that have such a control structure. Lisp is one. Certain varieties of productions systems also use recursive control. Unfortunately, I know of no way to formally specify a recursive control structure without also providing a fairly detailed specification of the language. For easy reference later, the basic idea will be recorded as an informal hypothesis, then a formal expression of it will be developed.

> *Recursive control structure*
> Procedures have the power of push down automata in that the representation of procedures permits goals to call themselves recursively, and the interpreter employs a goal stack.

There are several reasons for going beyond this informal expression and providing a formal description of the control structure. First, a formal description adds clarity not only to the description of the procedural representation but also to the other components of the model, such as repairs and deletion, which manipulate procedures and execution states. A second reason for the extra work of formalization is to bring up some tacit issues that are inherent in a recursive control regime. There are two such tacit issues. One concerns how the interpreter should choose which subgoal of a goal to run. The other concerns how the interpreter should decide when a goal is satisfied and may thus be popped from the goal stack.

To begin, a vocabulary is needed to speak of the static structure of procedures. For the purpose of control structure, only a few terms are needed:

goal:	A procedure has tokens (names) called goals.
rule:	The subgoals of a goal are represented as a set of rules "under" that goal.
applicability conditions:	The condition under which a subgoal may be executed are separated from it and used as the applicability condition of the corresponding rule.
action:	The other half of a rule is the name of the subgoal. In keeping with production system terminology, this is called the rule's action (or sometimes its subgoal).

When necessary for illustrations, the following syntax will be used:

Goal: Borrow-from
1. $T=0$ in the current column ⇒ BFZ
2. $T\neq0$ in the current column ⇒ Decrement

The goal is named Borrow-from. It has two rules. The rule numbers are used only as labels. Each rule has an applicability condition to the left of the arrow and an action (goal name in this case) to the right. Nothing important depends on this syntax.

A goal with no rules under it is called a primitive goal. Primitive goals are at the grain-size boundary of the procedural representation. They represent actions, such as moving the hand to write a digit, that are beneath the grain size of the model. In order to deal with them, the interpreter is equipped with a special operation, `Eval`, that may be applied to a primitive goal and the current runtime state. Such a call will change the external (problem) state, but it will not change the internal (execution) state. Managing the execution state is the interpreter's job, with some help from the local problem solver, of course.

Given the notion of goals, the internal (execution) state of the interpreter can be defined. It cannot simply be a stack of goals. It must have a little more information. The extra information is needed to execute conjunctive goals, such as Borrow, which perform several rules before popping. The extra information indicates which of the goal's rules have already been executed. This prevents the conjunctive goal from deciding to run the same rule over and over again. The easiest way to add this extra information to the execution state is to stipulate that the stack holds pairs consisting of (1) a goal, and (2) the subset of the goal's rules that have already been run. When a goal is first pushed onto the stack, the set of executed rules will, of course, be empty. There is nothing special going on here. Any recursive language's interpreter would have to have some such information (c.f., the refractoriness principle for conflict resolution in production systems, McDermott and Forgy, 1978).

There is one other kind of extra information that must be a part of the execution state. It is not a part of the stack. At minimum, only a single bit is needed. Basically, it remembers whether the interpreter's last action was a push or a pop. The reason this extra bit of control state is needed is that the Backup repair is impossible to implement without it. Backup pops the stack back to an OR goal (OR goals will be formally introduced in the next chapter). Normally, when control pops back to an OR, that goal immediately pops. Since OR goals only execute one rule, and a rule was just executed, the OR goal is done and should be popped. However, Backup needs to reset the execution state so that the OR will be resumed. The idea behind Backup is to take some alternate rule to the one that led to the impasse that it just repaired. The problem is how to tell the interpreter not to pop the OR but instead to pick another rule and execute it. Backup cannot change the set of tried rules associated with the goal. These must be left alone so that Backup will not cause the interpreter to take the impasse-causing rule over again. Since the stack will not suffice for Backup to communicate with the interpreter, a new bit of execution state is needed. This bit will be called *microstate.*

Formally, defining the interpreter means defining the function `Interpret`. In section 6.5, `Interpret` was introduced as an undefined function whose input is a runtime state and the core procedure. Its output is the next runtime state. Given the concepts introduced so far, it is easy to define `Interpret`. Some nomenclature is needed:

`(Goal-TOS S)`	The goal on the top of the stack in the runtime state S.
`(ExecutedRules-TOS S)`	The set of executed rules on the top of the stack of the runtime state S.
`(Microstate S)`	The microstate of the runtime state S.
`(Push S G RS)`	Changes the runtime state S by pushing the goal G and the rule set RS onto the stack. Also, it changes the microstate of S to Push.
`(Pop S)`	Pop's the runtime state's stack and sets its microstate to Pop.
`(Eval G S)`	Evaluates the goal G, which must be primitive. This changes only the external component (i.e., problem state) of runtime state S.

These are defined functions that access and change the runtime state in simple ways. However, two new functions are needed which will remain undefined until the next chapter:

`(PickRule S)`	Given the current runtime state S, `PickRule` chooses one of the top goal's rules as the next rule to execute and returns it. Typically, `PickRule` tests the applicability conditions of the rules of G that have not yet been executed. It finds the rules whose conditions are true in

	the current state. These are the applicable rules for the current instantiation of the goal. If several rules are applicable, `PickRule` applies criteria, called *conflict resolution strategies*, to chose which of the applicable rules to return. The next chapter specifies the conflict resolution strategies, thus defining the function.
`(ExitGoal? S)`	This predicate is true when the runtime state S is such that the current goal should be popped.

The definition of `Interpret` is shown in figure 9–4. There are two basic cases: (1) If microstate is Push, then the current goal has just been started up. If it is a primitive goal, then the interpreter just executes it; otherwise, a rule is chosen and executed. (2) If microstate is Pop, then the current goal has just had one of its rules executed. The choice is between resuming it, by choosing a rule and executing it, or exiting the goal by popping the stack.

Local problem solving and the execution state

As discussed in section 6.5, the local problem solver and the interpreter take turns examining the runtime state and modifying it. Although it is premature to venture a complete definition of how repairs and impasse conditions are implemented, it is interesting to sketch a few of them.

There is an impasse condition that checks the preconditions of goals. Preconditions need to be checked just before a goal is executed. This is easily done using microstate. Expressed informally, the impasse condition is a three-part conjunction:

1. `(Microstate S)=Push`, and
2. there exists a precondition C for `(Goal-TOS S)`, and
3. C is false in S.

Another impasse condition detects when the interpreter would halt because no rules apply. It's expression is also a conjunction with three conjuncts:

```
If (Microstate S)=Push then
   If (Goal-TOS S) is primitive then
     1. (Eval (Goal-TOS S) S)
     2. (Pop S)
   else (ExecuteRule S (PickRule S))
else
   If (ExitGoal? S) then (Pop S)
     else (ExecuteRule S (PickRule S)).

where
(ExecuteRule S R) ≡
   1.   Add R to (ExecutedRules-TOS S)
   2.   (Push S (ActionOf R) {})
```

Figure 9-4
Definition of `Interpret`.

1. `(Goal-TOS S)` is not a primitive goal, and
2. either `(Microstate S)=Push` or `(ExitGoal? S)=false`, and
3. `(PickRule S)` returns no rule.

The idea here is that the halt impasse condition first checks to see if `PickRule` will be called by the interpreter. If so, it calls `PickRule` itself to see if it is able to find a rule that is applicable. If not, then the procedure is stuck, the impasse condition is true, and a halt impasse occurs.

Repairs are equally simple to express. The Noop repair is virtually trivial. It simply calls `(Pop S)`. This simulates a return from the current goal. The Backup repair is only a little more complex. It calls `(Pop S)` until `(Goal-TOS S)` is a disjunctive goal that has some unexecuted rules. Then it sets microstate to Push. Since microstate is Push, the interpreter will wind up calling `PickRule` and executing another of the goal's rules. If Backup left microstate at Pop, then the interpreter would call `ExitGoal?` and probably wind up popping the stack that Backup so carefully adjusted. Microstate, or something like it, is crucial to expressing the Backup repair.

Chapter 10
Goal Types

The previous chapter dealt with control flow from a broad perspective. It contrasted three control regimes — finite state, recursive, and coroutine — that permeate the whole of the procedure representation language. There is another perspective on control flow, a more narrowly focused one, but one that is equally familiar to people designing or learning computer programming languages: what control constructions does the language allow? For instance, does the language support any of these:

if-then-else
Case or SelectQ
COND
AND
OR
While or Until loops
PROG or BEGIN...END Blocks
"For I from 1 to N do ...."
"For each x in the set S do ...."
"Find x in the set S such that ...."

The equivalent of such questions can be meaningfully asked of the language used to represent human procedures. Given that "goal" is the term used for a group of related subgoals, the question asks what *goal types* or *goal schemata* exist. For a procedure learning theory, postulating a goal type of the representation language means that the students have a prior expectation that a certain pattern of control will be common. If the learning theory postulates a bias (simplicity metric) of a procedure inducer, then hypothesizing a goal type amounts to parameterizing the bias so that the inducer tends to view examples as having a certain pattern of control, the one expressed by the goal type, rather than construing the examples as exhibiting other flows of control. Later, in the Bias level of this document, several inductive biases will be postulated. So the issue of goal types is an important one for step theory.

The same comments apply to repair theory. Both local problem solving and deletion are affected by the goal-subgoal hierarchies of the core procedures that they operate on. Since goal types can affect those structures, the existence and identity of goal types can impact repair theory.

There is, however, an inherent methodological difficulty in determining which goal types exist. Most goal types are redundant in that their pattern of control can be expressed without them, albeit less concisely. In fact, a single goal type suffices to express all the others. As proof, one can offer production systems. A typical production system uses just one goal type. For instance, a goal like the following one acts like a PROG (in Lisp) or a BEGIN-END block (in Algol):

Goal: Regroup (T)
1. {} ⇒ (Borrow-into T)
2. {} ⇒ (Borrow-from (Top-of-next-column T))

(In this example and the ones following it, a few inconsequential conventions have been adopted. Each goal has its subgoals listed as production rules beneath it, numbered for convenience only. The conditions governing when a rule can be executed are in braces on the left of the arrow. The rule's action is on the left. Arguments, such as T above, are treated as in Lisp or Algol. Rules are tested in order; the first one whose conditions are true is run, except rules that have been run already under the current invocation of the goal may not be run again. This convention corresponds to two common conflict resolution strategies for production systems, recency and refractoriness (McDermott & Forgy, 1978). Nothing in the following argument depends on the adoption of these conventions.) The goal above acts like a PROG because it executes both subgoals, Borrow-into and Borrow-from, and it does so in a fixed sequence. Other well-known goal types can also be emulated by production systems. A goal like:

```
Goal: BorrowFrom (T)
1.    {(Zero? T)} ⇒ (BorrowFromZero T)
2.    {(Not (Zero? T)) (Not (CrossedOut? T))} ⇒ (Decrement T)
```

acts like an if-then-else in that it only executes one of its subgoals depending on whether T is zero or not. The following goal acts like a loop:

```
Goal: Multi (C)
1.    {} ⇒ (Sub1Col C)
2.    {(Exists? (NextColumn C))} ⇒ (Multi (NextColumn C))
```

It does the "loop body," the subgoal Sub1Col, then tests for termination. It calls itself tail-recursively if it is not done yet. In short, a single goal type suffices to express many kinds of control.

All production systems that I know of use just one goal type. Exactly which goal type is used is different in different production systems since in production systems, the goal type's behavior depends on the production system's conflict resolution strategies (McDermott & Forgy, 1978, review a variety of conflict resolution strategies). Nonetheless, the principle of *homogeneous goal types* is so widely adhered to that it could even be taken as the defining characteristic of production systems.

Over the years, many procedures have been expressed in production systems. In some sense, this constitutes proof that the homogeneous goal type principle cannot be refuted on grounds of inexpressiveness. It does not limit the language so much that it becomes impossible to express some procedures. Consequently, any challenge to the homogeneous goal type principle will have to be made on more subtle grounds. As it turns out, the distinctions that will be used in this chapter are vanishingly subtle. For instance, it was mentioned that goal types affect inductive biases. Actually, they only affect the *elegance* of such biases. If the interpreter can distinguish different control flows, then so can the biases, although they might have to simulate execution of the procedure in order to do so. If students are biased towards forming loops, say, but the language doesn't have a loop goal type, then the theorist can write a bias that captures the students' predilections by, e.g., having the bias check for tail-recursive calling paths in the goal-subgoal hierarchy. This would make it an ugly, complicated bias, but it would capture the students' cognition. In short, the only way the existence of goal types will show up is in the *parsimony* of the theory. This is the methodological problem mentioned earlier. The goal type issue will be settled only by weak parsimony-based arguments.

This chapter considers three hypotheses about goal types. The first one is the homogeneous goal type principle. The second is the goal type principle that is used in the current version of the theory. The third is the goal type principle that was used in an early version of repair theory (Brown & VanLehn, 1980).

1. And: A goal is popped when all applicable subgoals have been tried. A subgoal is applicable if, e.g., the conditions on the left side of its rule are true.
2. And-Or: Goals have a binary type. If a goal's type is AND, all applicable subgoals are executed before the goal is popped. If it is OR, the goal pops as soon as one subgoal is executed.
3. Satisfaction conditions: Goals have a condition which is tested after each subgoal is executed. If the condition is true, the goal is popped. Metaphorically speaking, the goal keeps trying different subgoals until it is satisfied.

As it turns out, support for a fourth hypothesis has been recently discovered. The hypothesis extends the And-Or hypothesis by adding a third goal type, a loop across elements in a set, e.g., "Foreach x in the set S do ...." Evidence for it was discovered when Sierra traversed the Heath lesson sequence. This traversal, and the resulting core procedure tree, were discussed in section 2.8. Two branches of the core procedure tree (with suffixes P100 and Blk in figure 2–14), led to star bugs. The root of these mispredictions is the following: It appears that students can solve multi-column problems after seeing examples with at most two columns. But Sierra doesn't form the multi-column loop (actually, a tail recursion) until it is given three-column examples. For Sierra to form the loop on two-column examples would require biasing the learner in favor of loops. This is most easily done by giving the knowledge representation language a loop goal type. Although I won't recount more details here, it appears that the P100 star bugs would be avoided if (1) the procedure language had a Foreach goal type, and (2) Sierra's learner was biased towards it. Unfortunately, it requires much work to install this And-Or-Foreach hypothesis in Sierra, and I have not done so yet. Until that is done, there is no way to know whether the new hypothesis has some unfortunate side-effect that would cancel or outweigh its apparent benefits. Consequently, only the three hypotheses listed above will be given active consideration in this chapter.

There are two parts to the arguments of this chapter. The first part shows that the And-Or hypothesis is better than the And hypothesis. There are three separate arguments for this point, none of which is particularly convincing on its own. However, the fact that all of them point to the same conclusion provides support for the adoption of the And-Or hypothesis over its homogeneous cousin. The second part of the chapter is a competition between And-Or goal types and satisfaction conditions. It will be shown that satisfaction conditions provide better empirical coverage with respect to deletion, but they require unmotivated assumptions about learning. This argument is quite complex, so most of it has been moved to an appendix. Only a synopsis is presented in this chapter.

10.1 Assimilation is incompatible with the And hypothesis

One problem with the And hypothesis is due to the cumbersome way that disjunctive goals must be expressed. To express the fact that two subgoals are mutually exclusive, one must put mutually exclusive applicability conditions on them. For example, to express the fact that there are two mutually exclusive ways to process a column, depending on whether it's a two-digit column or a one-digit column, one would write:

```
Goal: process-column (C):
  1. (blank? (bottom C)) ⇒ (bring-down-top C)
  2. (not (blank? (bottom C))     ⇒ (take-difference C)
```

Both rules must have applicability conditions in order that they be mutually exclusive. On the problem

```
 37
- 4
```

the first rule must be prevented from applying to the units column, so its applicability condition is necessary. The second rule's applicability condition is necessary to prevent it from applying to the ten's column. Because the And hypothesis tries to execute *all* subgoals, one can only get mutual exclusion by using mutually exclusive applicability conditions.

This implies that assimilating a new alternative method of accomplishing a goal may involve rewriting the applicability conditions of the existing subgoals. If the applicability conditions are not changed, then the new subgoal will not turn out to be mutually exclusive of the old subgoal. For example, to assimilate a new method of processing columns, say one that handles columns whose top and bottom digits are equal (i.e., the rule N−N=0), one would have to modify the above goal to become:

```
Goal: process-column (C):
  1. (= (top C)(bottom C)) ⇒ (write-zero-in-answer C)
  2. (blank? (bottom C)) ⇒ (bring-down-top C)
  3. (and  (not (blank? (bottom C))
           (not (= (top C)(bottom C))) ⇒ (take-difference C)
```

Adding the new subgoal forced the applicability conditions of one of the existing subgoals to be changed (the underlined material was added). The essential point here is that the And hypothesis forces mutually exclusive alternatives to be highly interdependent. This lack of modularity forces learning to modify existing material even though that material's function has not changed. There are some potential drawbacks to this.

Mathematical skill acquisition is clearly incremental. New components of a skill are slowly added. Knowledge accrues, rather than springing full grown from some catalytic experience. Such plodding, slow-growing learning is often called *assimilation* to differentiate it from learning that takes the form of a radical restructuring of the student's knowledge. However, the hypotheses that have been accepted so far do not rule out such radical restructurings. In particular, the one-disjunct-per-lesson hypothesis rules out adding extra disjuncts but it says nothing about augmenting existing disjuncts, e.g., by adding conjuncts to applicability conditions, as in the example above. To capture the apparent quality of mathematical skill acquisition, the following hypothesis was used in an earlier version of the theory (VanLehn, 1983):

> *Assimilation*
> `Disjoin` only adds a new disjunct (subprocedure). It does not modify the existing knowledge structure in any other way.

This hypothesis says that subprocedure acquisition is an additive action. Learning doesn't change the old structure, it only adds a new chunk (disjunct, subprocedure). This hypothesis is yet another felicity condition on learning. It amounts to a guarantee to the students that what they learned earlier will remain valid and useful. To put it differently, the curriculum is arranged to be efficient. It doesn't teach a concept or a skill unless it will remain useful, possibly as the basis for further development of concepts or skills. As with the other felicity conditions, evidence in its favor is that current lesson sequences are constructed so that the correct procedure can be learned without violating the felicity condition. That, plus its inherent plausibility, are the the only known support. This is not sufficient, to my mind, so the assimilation hypothesis is not included in the current theory. Yet, if it were in the theory, as intuition dictates it should be, it would have entailments for the goal type controversy. The assimilation conjecture is incompatible with the And hypothesis. As we just saw, the And hypothesis forces too much of an existing core procedure to be changed in order to assimilate a new subprocedure. In short, to the extent that one accepts the assimilation conjecture, one must also reject the And hypothesis.

10.2 The And-Or hypothesis simplifies the Backup repair

There is a minor advantage to the And-Or hypothesis. Using both AND goals and OR goals simplifies the Backup repair. The function of the repair is to pop the goal stack back to the first goal that has some alternatives left to try. When goals are typed, it is trivial to tell whether a goal has any alternatives left. If it is an AND goal, by definition it does not. If it is an OR goal, then only one of its alternatives has been tried (because it normally pops after trying one subgoal), so all the rest must be open. Backup's search becomes trivial: pop the stack back to the first OR goal. That this simplicity falls out of the And-Or exit convention is weak evidence that the binary distinction is somehow a natural one for the procedural representation language to make.

10.3 Rule deletion needs the And-or distinctions

It was shown in chapter 7 that a certain group of bugs, labelled the deletion bugs, seem to require some kind of deletion operator in order for the theory to generate them. To generate them with incomplete learning and/or local problem solving would necessitate expanding the power of those mechanisms to an unacceptable degree.

This chapter picks up the deletion story and considers how deletion can be formalized. The first step is to show that simply deleting rules will suffice to generate the deletion bugs. The next step will be to show that not all rules should be subject to deletion. If certain rules are deleted, then star bugs are generated. To avoid such absurd predictions, some constraints must be placed on the rule deletion operator. This is where the goal type controversy comes in. It can be shown that the rules that should not be deleted are exactly the ones under disjunctive goals. Once again, the distinction between AND goals and OR goals arises naturally. If the representation uses a binary type to distinguish the two goal types, then the deletion operator becomes trivially simple. Without it, the operator must examine the applicability conditions of rules to infer which goals are disjunctive ones. The fact that a simple statement of the deletion operator falls out under the And-Or hypothesis is more converging evidence that it is the right one for the knowledge representation to employ.

In order to discuss how to generate the deletion bugs, it helps to have a concrete expression of a subtraction procedure so that deletion may act upon it. Figure 10–1 shows a particular subtraction procedure, and gives a brief explanation of it. The procedure is expressed in a rule-oriented syntax. Since the issue of goal types is still under discussion, and the applicability conditions depend on the exit convention in use, the expression of figure 10–1 uses an informal representation for applicability conditions in order to stay neutral.

Given this particular procedure for subtraction, some general points about deletion can be made. For illustration, suppose that deletion is formalized as deleting a rule, any rule. Since there are 18 rules in the procedure, there are 18 possible deletions. (This is not quite a straw man, by the way, since it can be taken as one formulation of Young and O'Shea's approach (1981) to generating bugs.) The first point to make is that rule deletion does indeed generate the set of deletion bugs. Deleting rule 11 generates the following deletion bug:

Goal: Sub (P)
1. P is a multi-column problem ⇒ (Multi (Rightmost-column P))
2. P is a single-column problem ⇒ (Diff (Rightmost-column P))

Goal: Multi (C)
3. true ⇒ (Sub1Col C)
4. true ⇒ (Sub-rest (Next-column C))

Goal: Sub-rest (C)
5. C is not the leftmost column ⇒ (Multi C)
6. C is the leftmost column, and its bottom is blank ⇒ (Show C)
7. C is the leftmost column, and its bottom is not blank ⇒ (Diff C)

Goal: Sub1Col (C)
8. T<B in C ⇒ (Borrow C)
9. the bottom of C is blank ⇒ (Show C)
10. C is normal: T≥B and its bottom is not blank ⇒ (Diff C)

Goal: Borrow (C)
11. true ⇒ (Borrow-from (Next-column C))
12. true ⇒ (Add10 C)
13. true ⇒ (Diff C)

Goal: Borrow-from (C)
14. T=0 in C ⇒ (BFZ C)
15. T≠0 in C ⇒ (Decrement-top C)

Goal: BFZ (C)
16. true ⇒ (Borrow-from (Next-column C))
17. true ⇒ (Add10 C)
18. true ⇒ (Decrement-top C)

The Sub goal simply chooses between trivial, one-column processing and the usual multiple-column procedure. The two goals, Multi and Sub-rest, express a loop across columns as a tail recursion. Next-column is a function that takes a column and returns the next column to the left. If the column C that is given to Sub-rest is the leftmost column, Sub-rest answers it with either rule 5 or rule 6, thus terminating the recursion. Show and Diff are primitives. Show writes the top digit of a column as its answer. Diff takes the column difference. Columns other than the leftmost column are answered by Sub1Col. Sub1Col is the main column processing goal. If the top digit of the column is less than the bottom digit (i.e., T<B in the shorthand used throughout this document), then Sub1Col calls borrowing (rule 8). If the column has a blank instead of a bottom digit, then it simply writes the top digit as the answer (rule 9). Otherwise, it does the usual take-difference operation. The Borrow goal first borrows from the next column to the left, then adds ten to the column originating the borrow. Add10 is a primitive that adds ten to the top digit of the column its given. Borrow winds up by taking the difference in the original column. There are two ways to achieve the Borrow-from goal. If the column's top digit is non-zero (rule 15), then the procedure simply decrements it by one (Decrement-top is a primitive). If the column's top digit is zero, then Borrow-from calls BFZ. The goal BFZ first borrows from the next digit to the left (i.e., if the borrow originated in the units, this would be a borrow from the hundreds column), then adds ten to the current column. Since the top digit was zero, this means the top digit will become a ten. The next rule, 18, decrements this ten to a nine. This finishes the BFZ.

Figure 10–1

A subtraction procedure, presented in an informal rule-oriented syntax, with explanation.

Borrow-No-Decrement:

$$\begin{array}{rl} 3\;4\;5 & \\ -\,1\;0\;2 & \\ \hline 2\;4\;3 & \checkmark \end{array} \qquad \begin{array}{rl} 3\;4^{1}5 & \\ -\,1\;2\;9 & \\ \hline 2\;2\;6 & \times \end{array} \qquad \begin{array}{rl} 3^{1}0^{1}7 & \\ -\quad 6\;9 & \\ \hline 3\;4\;8 & \times \end{array}$$

This bug does only the borrow-into half of borrowing. It omits the borrow-from half. It is a deletion bug because its familiarity with borrowing makes it likely that the students with this bug have been taught simple borrowing. However, part of borrowing apparently did not sink in or it was forgotten. Another deletion bug results from deleting rule 16.

Borrow-From-Zero:

$$\begin{array}{rl} 3\;4\;5 & \\ -\,1\;0\;2 & \\ \hline 2\;4\;3 & \checkmark \end{array} \qquad \begin{array}{rl} 3\;\overset{3}{4}{}^{1}5 & \\ -\,1\;2\;9 & \\ \hline 2\;1\;6 & \checkmark \end{array} \qquad \begin{array}{rl} 3^{1}\overset{9}{0}{}^{1}7 & \\ -\,1\;6\;9 & \\ \hline 2\;3\;8 & \times \end{array}$$

This bug only does part of borrowing across zero. It changes the zero to ten then to nine, but does not continue borrowing to the left. Because it does do part of borrowing across zero, it is likely that subjects with this bug have been taught borrowing across zero. It is also clear that they did not acquire all of the subprocedure, or else they forgot part of it. If the subtraction curriculum was such that teachers first taught one half of borrowing across zero, and some weeks later taught the other half, then one would be tempted to account for this bug with incomplete learning. But borrowing from zero is, in fact, always taught as a whole. So some other formal technique, deletion, is implicated in this core procedure's generation.

A third deletion bug is generated by deleting rule 18. It seems to forget to do the second decrement in the borrow across zero:

Don't-Decrement-Zero:

$$\begin{array}{rl} 3\;4\;5 & \\ -\,1\;0\;2 & \\ \hline 2\;4\;3 & \checkmark \end{array} \qquad \begin{array}{rl} 3\;\overset{3}{4}{}^{1}5 & \\ -\,1\;2\;9 & \\ \hline 2\;1\;6 & \checkmark \end{array} \qquad \begin{array}{rl} \overset{2}{3}{}^{1}0^{1}7 & \\ -\,1\;6\;9 & \\ \hline 1\;7\;8 & \times \end{array} \qquad \begin{array}{rl} \overset{2}{3}{}^{1}0^{1}7 & \\ -\qquad 9 & \\ \hline 2\;10\;8 & \times \end{array}$$

This shows up rather clearly in the last problem. The zero has had a ten added to it, but it has not been decremented as it should be to complete the borrow from zero. Consequently, the answer in the tens column is a two digit number. (This sometimes triggers a second impasse.) A more detailed description of Don't-Decrement-Zero appears in section 7.1.

There are other deletion bugs, but these three suffice to show that rule deletion is a productive addition to the theory. These three bugs will be used as a yardstick to measure the empirical adequacy of the various kinds of deletion that will be discussed.

Unconstrained rule deletion overgenerates

The problem with unconstrained rule deletion is that it overgenerates. About half the rule deletions generate star bugs. Such star bugs must be blocked if the theory is to have any empirical worth at all. The issue is how to constrain rule deletion. A birds-eye view of the issue is provided by figure 10-2, which lists each rule along with what its deletion results in (roughly). Some deletions cause star bugs, some cause observed bugs, and some cause bugs that have not yet been observed but are plausible predictions for future observations.

Goal: Sub
1. predicted bug: only does single column problems.
2. *star* bug: can't do single column problems but does others perfectly.

Goal: Multi
3. predicted bug: only does the leftmost column.
4. *star* bug when BFZ exists: does units column only, but perfectly, even if BFZ is required.
   predicted bug if Borrow not yet learned: does units column only, taking absolute difference.

Goal: Sub-rest
5. predicted bug: only does leftmost column.
6. predicted bug: forgets leftmost column when it has a blank bottom.
7. *star* bug: leaves leftmost column unanswered unless it has a blank bottom.

Goal: Sub1Col
8. various observed bugs: e.g., Smaller-From-Larger, Zero-Instead-of-Borrow.
9. various observed bugs: e.g., Quit-When-Bottom-Blank, Stutter-Subtract.
10. *star* bug: does borrow columns but leaves ordinary columns unanswered.

Goal: Borrow
11. observed bug: Borrow-No-Decrement.
12. a *star* bug, Blank-With-Borrow-From, and various observed bugs, e.g., Smaller-From-Larger-With-Borrow.
13. *star* bug: does scratch marks for borrowing, but leaves the column unanswered.

Goal: Borrow-from
14. various observed bugs: e.g., Stops-Borrow-At-Zero.
15. *star* bug: never does the leftmost decrement of borrow, including BFZs.

Goal: BFZ
16. observed bug: Borrow-From-Zero.
17. a *star* bug, Blank-With-Borrow-Across-Zero, and various observed bugs e.g., Borrow-Across-Zero.
18. observed bug: Don't-Decrement-Zero.

Figure 10–2

Results from each rule deletion of the subtraction procedure, indicating which generate star bugs.

Inspection of figure 10–2 reveals that the deletions that generate star bugs fall into two basic groups. Half of the star deletions are rules beneath disjunctive goals (deletions of rules 2, 7, 10 and 15), or rather goals that function as OR goals even though they would not be marked as such under the And exit convention. The remaining star bugs (rules 4, 12, 13 and 17) have the characteristic that they know how to borrow from zero, indicating some sophistication in subtraction, but they nonetheless leave certain columns unanswered. The juxtaposition of such sophistication in borrowing with missing knowledge about answering columns makes these bugs highly unlikely. In the next section, these star bugs will be discussed. In this section, attention will be focused on blocking the star bugs of the first group

Deleting any of rules 2, 7, 10 or 15 generates a star bug. These rules are all beneath disjunctive goals. Deletion of the sisters of these rules (i.e., rules 1, 5, 6, 8, 9, or 14) generate bugs, some of which are observed, but they are all bugs that would be generated by incomplete traversal of the lesson sequence. Deletion under a disjunctive goal hurts the theory's empirical adequacy if it affects it at all. To drive this point home, consider the Borrow-from goal. It has two rules. Deleting the second one, rule 15, hurts the theory. Rule 15 does borrowing from non-zero digits.

It decrements the digit by one. If rule 15 is deleted, a star bug occurs. Some of the star bug's work is illustrated below:

*Only-Borrow-From-Zero:

$$\begin{array}{r} 3\,4\,5 \\ -\,1\,0\,2 \\ \hline 1\,4\,3 \end{array}\ \checkmark \qquad \begin{array}{r} 3\,4^{1}5 \\ -\,1\,2\,9 \\ \hline 2\,2\,6 \end{array}\ \times \qquad \begin{array}{r} 2\,\overset{9}{0}{}^{1}7 \\ -\,1\,6\,9 \\ \hline 1\,3\,8 \end{array}\ \times$$

This star bug misses all problems requiring borrowing because it never performs a decrement, despite the fact that it shows some sophistication in borrowing across zero in that it changes zeros to nine. The juxtaposition of this competency in borrowing across zero with missing knowledge about the simple case makes the bug highly unlikely.

The other rule of the goal Borrow-from, rule 14, does borrowing for zeros. It simply calls the BFZ goal. If rule 14 is deleted, the procedure acts just like BFZ had never been taught. In particular, the deletion would generate the bug Stops-Borrow-At-Zero, which has been used as a prime example of the combination of incomplete learning and local problem solving (see section 2.9). Blocking the deletion of rule 15 is good because it prevents generation of a star bug. Blocking deletion of rule 14 doesn't hurt because its bugs have alternative derivations. The point is this: If all rule deletions beneath Borrow-from are banned, the theory's predictions are improved.

The results of figure 10–2 clearly indicate that rule deletion should not apply to rules beneath goals, such as Borrow-from, that are disjunctive in nature. Only rules beneath conjunctive goals should be subject to deletion. Although the results of figure 10–2 are, of course, sensitive to the particular structure used in the procedure of figure 10–1, the restriction on rule deletion has been tested on Sierra with other procedures and found to hold up just as well. Although the restriction still allows some star bugs to be generated, it blocks the generation of many others. Hence, there is strong evidence that deletion should be constrained to delete only rules beneath goals that are conjunctive in nature.

Conclusions

Once again, the distinction between AND goals and OR goals has arisen from trying to fit an operator around the empirical evidence. The deletion operator needs the And-Or distinction, just as the Backup repair did. It seems that nature is trying to tell us something. The And-Or distinction seems a fundamentally important distinction, and as such should be given a clear expression in the representation language, rather than lurking in the rule's applicability conditions in the form of mutually exclusive predicates. On this basis, the And hypothesis will be eliminated from further consideration. Goals will have at least a binary type, AND versus OR, and rule deletion will be limited to rules that appear beneath AND goals.

10.4 Satisfaction Conditions

Conjunctive rule deletion generates all the deletion bugs and it avoids generating half of the star bugs of figure 10–2. However, it still allows the other half of the star bugs to be generated. These star bugs will be examined in detail in order to motivate a way of blocking their deletion.

The main loop of subtraction, which traverses columns, has the following goal structure when it is translated into the And-Or exit convention from the neutral representation of figure 10–1:

Goal: Multi (C) Type: AND
3. (Sub1Col C)
4. (Sub-rest (Next-column C)

Goal: Sub-rest (C) Type: OR
5. C is not the leftmost column ⇒ (Multi C)
6. C's bottom is blank ⇒ (Show C)
7. true ⇒ (Diff C)

The applicability conditions of the rules have been adjusted. The conditions for the AND rules have been omitted. The conditions for the OR goal, Sub-rest, have been adjusted to reflect the fact that they are tested in order and only one is executed. For instance, if the column C is the leftmost column, then rule 5 will not be applicable, and control moves on to test rule 6. If that rule applies, the primitive Show answers the column, then control returns to Sub-rest. Since the goal is marked as an OR goal, and one rule has been executed, no more rules are tested. In particular, the default rule, 7, will not be tested. Instead, the Sub-rest goal is popped.

The Multi goal is an AND goal, so either of its subgoals can be deleted. Deleting rule 4 creates a bug that only does the units column. Intuitively, only doing one column would be the mark of a student who has not yet been taught how to do multiple columns. Since doing multiple columns is always taught before borrowing, it would be highly unlikely for a student to know all about borrowing and yet do only the units column. To put it more formally, if all of BFZ were present when rule 4 is deleted, the procedure would generate a star bug:

Only-Do-Units:

$$\begin{array}{r} 3\,4\,5 \\ -\,1\,0\,2 \\ \hline 3\ \times \end{array} \qquad \begin{array}{r} 3\,\overset{3}{4}{}^{1}5 \\ -\,1\,2\,9 \\ \hline 6\ \times \end{array} \qquad \begin{array}{r} \overset{2}{3}{}^{1}\overset{9}{0}{}^{1}7 \\ -\,1\,6\,9 \\ \hline 8\ \times \end{array}$$

If borrowing were not yet learned and rule 4 were deleted, then reasonable bugs would be generated. For instance, one reasonable, but as yet unobserved, buggy procedure does only the units column but it simply takes the absolute difference there instead of borrowing. It would be the bug set {Only-Do-Units Smaller-From-Larger}. In short, there is nothing wrong with the deletion of rule 4 *per se*, but it can create a procedure that mixes competence with incompetence in an unlikely manner.

Another star bug of figure 10–2 occurs when rule 13 is deleted from Borrow, given the following version of column processing and borrowing:

Goal: Sub1Col (C) Type: OR
8. T<B in C ⇒ (Borrow C)
9. the bottom of C is blank ⇒ (Show C)
10. true ⇒ (Diff C)

Goal: Borrow (C) Type: AND
11. (Borrow-from (Next-column C))
12. (Add10 C)
13. (Diff C)

Deleting rule 13 generates a procedure that sets up to take the column difference after a borrow, but forgets to actually take it. This leads to the following star bug:

*Blank-With-Borrow:

$$\begin{array}{rrr} & & \\ 3&4&5 \\ -1&0&2 \\ \hline 2&4&3 \end{array}\ \checkmark \qquad \begin{array}{rrr} &3& \\ 3&4&{}^{1}5 \\ -1&2&9 \\ \hline 2&1& \end{array}\ \times \qquad \begin{array}{rrr} 2&9& \\ 3&0&{}^{1}7 \\ -1&6&9 \\ \hline 1&3& \end{array}\ \times$$

What makes this bug so unlikely is that it leaves a blank in the answer despite the fact that it shows a sophisticated knowledge of borrowing.

It is possible to put explicit constraints on conjunctive rule deletion in order to block the deletions that generate the star bugs. However, there is another way to prevent overgeneration that will be shown to have some advantages. The basic idea is to make the operator inapplicable by changing the types of the two goals in question so that they are not AND goals. This would make the deletion operator inapplicable. That is, one changes the knowledge representation rather than the operator.

The proposed change is to adopt a new goal type. The hypothesis is to generalize the binary AND/OR type to become *satisfaction conditions.* The basic idea of an AND goal is to pop when *all* subgoals have been executed, while an OR goal pops when *one* subgoal has been executed. The idea of satisfaction conditions is to have a goal pop *when its satisfaction condition is true.* Subgoals of a goal are executed until either the goal's satisfaction condition becomes true, or all the applicable subgoals have been tried. (Note that this is not an iteration construct — an "until" loop — since a rule can only be executed once.) AND goals become goals with FALSE satisfaction conditions: Since subgoals are executed until the satisfaction condition becomes true (which it never does for the AND) or all the subgoals have been tried, giving a goal FALSE as its satisfaction condition means that it will always execute all its subgoals. Conversely, OR goals are given the satisfaction condition TRUE: The goal exits after just one subgoal is executed.

With this construction in the knowledge representation language, one is free to represent borrowing in the following way:

```
Goal: Sub1Col (C) Satisfaction condition: C's answer is non-blank:
   8.  T<B in C ⇒ (Borrow C)
   9.  the bottom of C is blank ⇒ (Show C)
   10. true ⇒ (Diff C)

Goal: Borrow (C) Satisfaction condition: false
   11. (Borrow-from (Next-column C))
   12. (Add10 C)
```

The AND goal, Borrow, now consists of two subgoals. After they are both executed, control returns to Sub1Col. Because Sub1Col's satisfaction condition is not yet true — the column's answer is still blank — another subgoal is tried. Diff is chosen and executed, which fills in the column answer. Now the satisfaction condition is true, so the goal pops.

Given this encoding of borrowing, the conjunctive rule deletion operator does exactly the right thing when applied to Borrow. In particular, since rule 13 is no longer present, it is no longer possible to generate the star bug, *Blank-With-Borrow-From by deleting it. Rule 13 has been merged, so to speak, with rule 10. Since rule 10 is under a non-AND goal, Sub1Col, it is protected from deletion.

Similarly, the star bug associated with column traversal can be avoided by restructuring the loop across columns. The two goals, Multi and Sub-rest, are replaced by a single goal:

Goal: SubAll (C) Satisfaction condition: C is the leftmost column.
5. true ⇒ (Sub1Col C)
6. true ⇒ (SubAll (Next-column C))

This goal first processes the given column by calling the main column processing goal, Sub1Col. Then it checks the satisfaction condition. If the column is the problem's leftmost column, the goal pops. Otherwise, it calls itself recursively. By using a satisfaction condition formulation, generation of the star bug is avoided. The AND goal, Multi, has been eliminated along with its rule 3, the rule whose deletion caused the star bug.

These two illustrations indicate that augmenting the representation with satisfaction conditions creates an empirically adequate treatment of deletion. Satisfaction conditions were used for several years in Sierra (Brown & VanLehn, 1980; VanLehn, 1983). However, as the first versions of Sierra's learner were implemented, a fatal flaw was discovered. In essence, if the learner was constructed so that it would put satisfaction conditions on Sub1Col and SubAll, it would also put satisfaction conditions on other goals, which, unfortunately, would block the generation of certain deletion bugs. This hurts the empirical adequacy of the theory. Various ad hoc constraints can be imposed to "fix" the flaw, but they, in turn, cause further empirical difficulties. The whole story is quite complex, so it has been relegated to appendix 10.

As shown there, a simpler solution is to abandon satisfaction conditions entirely. Instead, an explict constraint is added to the deletion operator: It may only delete rules from *the most recently acquired* AND goal. To see how this works, consider the star bug mentioned earlier, Only-Do-Units. This bug is an unreasonable prediction precisely because it exhibits perfect knowledge of borrowing, but has, apparently, "forgotten" that all columns need answering. To generate this star bug, a certain rule of the goal Multi (see above) must be deleted. It cannot be deleted *after* borrowing-from-zero is learned, because that is precisely what the new constraint blocks. But the deletion won't survive if it occurs before borrow-from-zero is learned. To see this, suppose it were deleted before then. The learner would try to use the damaged procedure to parse the borrow-from-zero examples. Even the simplest borrow-from-zero examples have at least two columns that require answers. Because of the rule deletion, the learner is unable to match the actions of the examples that answer the non-units columns. Of course, the learner is also unable to match the new borrow-from-zero subprocedure, which is the topic of the lesson. To assimilate the example, the learner would have to adjoin *two* disjuncts to the procedure — one to handle the extra answer actions, and one to handle the new subprocedure. Adding two disjuncts is prohibited by one-disjunct-per-lesson. In short, the damaged procedure cannot be augmented with the borrow-from-zero subprocedure. The case presented in this illustration is a typical one. Rule deletions that occur early in the lesson sequence generally will not survive long. A damaged procedure will be asked to take a lesson that assumes it has a subprocedure that it doesn't have, and normal, one-disjunct-per-lesson learning will not let this procedure pass. (In real classrooms, what probably happens is that students who have such damaged core procedures are discovered and remediated.) Hence, the only way to inappropriately juxtapose incompetency with competency is to delete rules from subprocedure well after they are acquired. This way of generating star bugs is exactly what the new constraint blocks. Only the AND rules of newly acquired subprocedures may be deleted.

10.5 Summary, formal hypotheses and conflict resolution

The goal type issue has proved to be a subtle one. The alternative exit conventions are fairly clear cut:

1. AND: A goal is popped when all subgoals have been tried.
2. AND/OR: Goals have a binary type. If the goal's type is AND, all applicable subgoals are executed before the goal is popped. If it is OR, the goal pops as soon as one subgoal is executed.
3. Satisfaction conditions: Goals have a condition which is tested after each subgoal is executed. If the condition is true, the goal is popped. Metaphorically speaking, the goal keeps trying different subgoals until it is satisfied.

However, the arguments between them were weak. What evidence there is indicates that the And-Or exit convention is the best. A brief review of the arguments follows.

There were three arguments against the And hypothesis. Two were based on parsimony. In order to have a simple expression of the Backup repair and conjunctive rule deletion, goals should be marked with a binary type to differentiate AND goals from OR goals. Under the And hypothesis, it is still possible to express the operators, but they must analyze the applicability conditions on rules in order to distinguish AND goals from OR goals. The third argument against the And hypothesis relies on intuition. Intuition, but little else, supports a conjectured felicity condition called the *assimilation conjecture*. It states that new knowledge structures can be acquired without changing the old ones, except in certain narrowly prescribed ways. Since the And hypothesis uses mutually exclusive applicability conditions to express disjunctive goals, it forces old applicability conditions to be adjusted when a new one is added. The And hypothesis forces learning to violate the assimilation conjecture.

Satisfaction conditions are a generalization of the And-Or convention. Under the And-Or convention, a goal pops after either one or all of its subgoals are executed. With satisfaction conditions, a goal may pop after any number of its subgoals have been executed, where the number of subgoals executed is governed by a condition. The extra degrees of freedom in expressive power can be used to control the predictions. Indeed, an attempt to do this was made by using satisfaction conditions to control the deletion operator. If satisfaction conditions were based on just the right goals, the theory avoided generating several pesky star bugs. However, it turned out to be difficult to account for the acquisition of satisfaction conditions on just these goals and not the others. The satisfaction conditions approach is weakened because learning cannot explain their existence. This leaves the And-Or hypothesis the only one standing.

Formal hypotheses

The main results of this chapter are summarized in the following three hypotheses:

And-Or
Goals bear a binary type. If G is the current goal in runtime state S, then
`(ExitGoal? S)` is true if
1. G is an AND goal and all its rules have been executed, or
2. G is an OR goal and at least one of its rules have been executed.

AND rule deletion
`(Delete P)` returns a set of procedures P′ such that P′ is P with one or more AND rules deleted.

Most recent rule deletion
`(Delete P)` returns a set of procedures P′ such that P′ is P with one or more rules deleted from the most recently acquired subprocedure.

These hypotheses define two previously undefined functions, `Delete` and `ExitGoal?`. The latter was used in the definition of the interpreter in section 9.4. It controls whether a goal is popped, given that the stack has just popped back to it.

Conflict resolution

The other undefined function used in the interpreter is `PickRule`. Its definition is simple to motivate. However, it depends on the And-Or distinction, which is why its definition has been deferred until now.

The function `PickRule` decides which of the current goal's rules the interpreter should run next. This choice is governed by conventions that are called *conflict resolution strategies* in the production system literature (McDermott & Forgy, 1978). Two conventions are needed just to get the interpreter to work at all:

1. If a rule has already been executed for this instance of the goal, then it may not be executed again.
2. If the applicability condition of a rule is false, it may not be run.

These two conventions were discussed in section 9.4. However, they do not always settle the question of which rule to pick. There are often several unexecuted, applicable rules for the current goal. More conventions are needed. It is convenient to discuss conventions for AND goals separately from the conventions for OR goals

AND goal conflict resolution

The problem with AND rules is expressing their sequential order. One way to get AND rules to run in sequence is to use the applicability conditions. In a production system, one can force rules to be executed in sequence by having each rule add a token to the working memory (execution state) that will trigger its successor rule and only its successor rule. This will not work here because the only internal state is the stack and the microstate bit. There is no buffer to add tokens to. On the other hand, the applicability condition could sense the external state (i.e., what the problem looks like). This would suffice for sequential ordering in many cases. However, it would interact with rule deletion in such a way that several of the deletion bugs could not be generated. I won't go through the details here. The point is that applicability conditions cannot be used to express the sequential order of AND rules. Some explicit convention is needed. About the simplest one possible is to represent AND rules in an ordered list, and to execute the rules in the order that they appear in the AND goal's list. It is the one that will be adopted.

OR goal conflict resolution strategies

The learner must induce the applicability conditions of OR rules. In particular, a new subprocedure will have a new OR rule that calls it. This OR rule is called the adjoining rule because it attaches the new subprocedure to the existing procedural structure. In order to induce the applicability condition of the adjoining rule, it is extremely helpful for the learner to be presented with negative instances of it. A negative instance of an applicability condition is a problem state where the applicability condition is false. Teachers and textbooks do not usually show negative instances to the students explicitly (i.e., in discrimination examples). However, the learner can recover negative instances from the kinds of examples they do get, under certain circumstances. Suppose the new subprocedure is a sister of some goal G. That is, G and the new subprocedure are both subgoals of the same OR goal. Given a worked example, the learner can discover whether

G has been executed and if so, what the problem state was at the time is was invoked. Call this problem state S. Since G and the new subprocedure are OR-sisters, the new subprocedure could have been picked to run at S, but the teacher did not do so. Because the new subprocedure was not run at S, its adjoining rule's applicability condition must have been false. Therefore, S is a negative instance for it. S is just what induction needs.

However, there is a subtle flaw in the inference just given. The choice of rule depends both on applicability conditions and on conflict resolution strategies. The learner cannot be sure that the adjoining rule's applicability condition is false at S unless the learner knows that the adjoining rule would be favored over the others in cases where they were both true. The only way to guarantee this is to adopt a conflict resolution strategy that is based on the time of acquisition. That is, when two rules are both applicable, the rule that was learned most recently is chosen. This convention, called "recently in long-term memory" when used in production systems, guarantees that the adjoining rule would have been chosen if it had been applicable: Since it wasn't chosen, it must not have been applicable. Therefore S is a valid negative instance for the induction of the adjoining rule's applicability condition.

In short, there is really no choice about conflict resolution strategies, given that goals are typed. The conflict resolution strategies can be summed up with the following hypothesis:

Conflict resolution

1. A rule may be executed only if its applicability condition is true and it has not yet been executed for this instance of the current goal.
2. In the representation of the procedures, the rules of AND goals are linearly ordered. If the current goal is an AND goal, and there are several unexecuted, applicable rules, then choose the first one in the goal's order.
3. If the current goal is an OR goal, and there are several unexecuted, applicable rules, then choose the rule that was acquired most recently.

Clauses 2 and 3 imply that AND goals and OR goals have similar syntax. They both have ordered lists of rules. Moreover, the interpreter always picks the first unexecuted rule, for AND goals, and the first unexecuted rule whose applicability condition is true, for OR goals. The major difference is what happens after the respective rules are executed. OR goals immediately pop; AND goals do not.

Chapter 11
Data Flow

This chapter discusses how the procedure representation language should represent data flow. In general, data flow in a procedure is the set of mechanisms and conventions for the storage and transmission of data, such as numbers or other symbols. (N.B., I am using "data flow" in the sense of Rich and Shrobe (1976) rather than the more common usage of Dennis (1974) and other authors who discuss data flow languages.) In the kinds of procedures that concern this theory, the data flow issue hinges mostly on how the language should represent focus of visual attention. As students solve subtraction problems, they seem to focus their visual attention on various digits or columns of digits at various times. This can be inferred from eye tracking studies (Newell & Simon, 1972; Buswell, 1926). It can also be inferred from the information that students read from the paper. The issue this chapter discusses is how to represent the fact that focus of attention is held unchanged for periods of time as well as being shifted. To put it in slightly inaccurate terms, the issues concern the short-term storage of visual focus. Four positions will be contrasted:

1. *No data flow*: The hypothesis is that procedures do not use data flow. In particular, there is no internal storage of focus of attention. Instead, the places where the procedure reads and writes on the page are described by static descriptions. For instance, instead of describing the place to write an answer digit as "the answer position in the *current* column," it would be described as "just to the left of the leftmost digit in the answer row." This description does not use the notion of a current focus of attention. It describes locations statically.

2. *Globally bound data flow (registers):* A leading contender for storing and transmitting focus of attention is to use registers that store either the position on the paper that is currently being examined, or a focus of attention that was once current and is being saved for some reason. By "register," I will mean a *globally* bound data storage resource. The current contents of a globally bound register is determined solely by chronology. Its contents is the value most recently set into the register.

3. *Locally bound data flow (schema-instance):* To describe this hypothesis, the notion of an instantiation of a goal is needed. When a goal is called, it is pushed onto the goal stack along with a little extra information. This extra information is labelled an *instantiation* of the goal. If a goal is called recursively, there will be two instantiations of it on the stack. The basic idea of the schema-instance data flow hypothesis is that data can be stored with each instantiation of a goal. The goal is viewed as a schema with certain open parameters, called arguments. Instantiating a goal fills in the values of those open parameters, an operation called binding the arguments. This way of structuring data flow is called *local* binding in recursive programming languages, such as Lisp. In object-oriented languages, such as Smalltalk, the same idea is used a little differently. In order to include object-oriented languages in the purview of this hypothesis, the hypothesis' name is schema-instance instead of the less general name, local binding.

4. *Applicative data flow:* The schema-instance data flow hypothesis allows different instantiations of a goal to have different foci of attention stored with them. However, it is possible for an instantiation of a goal to change its stored focus and even to change the stored foci of other goals' instantiations. The applicative data flow hypothesis outlaws such changes. Once an instantiation's arguments are bound, they can never be changed. Applicative data flow is used by applicative languages, such as pure Prolog or the lambda calculus.

The basic issue behind these various options concerns how independent data flow should be from control flow. The discussion starts with the weakest, most inflexible hypothesis: The no-data-flow hypothesis is that data flow is congruent with the *static* structure of the procedure. Thus, focus of attention doesn't depend on which instantiation of the goal is executing, but only on the name of the goal that is executing. This turns out to be too inflexible. It can't express certain bugs.

The next strongest hypothesis is the applicative hypothesis. It assumes that data flow is congruent with the *dynamic* structure of the procedure. If there is a stack, the focus of attention changes when, and only when, the stack is pushed or popped. This turns out to be the best of the four alternative hypotheses.

The global binding hypothesis allows data flow and control flow to be totally independent. The procedure is allowed to change the focus of attention without changing the flow of control, and vice versa. It is shown that this independence leads to problems in the theory. In particular, certain shifts in control necessitate a shift in data flow. In such cases, mainly concerning popping the stack, the independence of control and data flow must be curtailed: data flow must parallel control flow then. Such cases motivates the schema-instance hypothesis.

The schema-instance hypothesis is halfway between the total independence of control flow and data flow, and the total isomorphism of the two that is stipulated by the applicative hypothesis. Under the schema-instance hypothesis, when control flow changes, data flow changes. If there is a stack, focus automatically shifts when the stack changes because the top of the stack is what holds the current focus of attention. However, data flow may also be changed when control flow does not change. This extra degree of freedom is never used in any of the procedures implicated by the data. To explain this, a constraint is added: when the stack does not change, focus of attention does not change. The result is the applicative data flow hypothesis, that data flow can change when, and only when, control flow changes.

11.1 The hypothesis that there is no data flow

It may be that there is no need to have an explicit representation for data flow. Maybe it will suffice just to have a push-down automaton, not an ATN with its registers. This would put a burden on the procedure's interface with the problem state. In order to traverse columns in subtraction, instead of passing the current column in a register, the patterns (or whatever implements the interface) would have to describe the focus of attention as the rightmost unanswered column. Since there is a visual marker for where focus of attention needs to be, namely the boundary between answered and unanswered columns, this technique will succeed.

However, there are bugs which leave answers blank. These cannot be represented by using the boundary between answered and unanswered columns. For instance, one observed bug skips columns which require borrowing:

Blank-Instead-of-Borrow:

$$\begin{array}{r} 3\,4\,5 \\ -\,1\,0\,2 \\ \hline 2\,4\,3 \end{array}\ \checkmark \qquad \begin{array}{r} 3\,4\,5 \\ -\,1\,2\,9 \\ \hline 2\,2\, \end{array}\ \times \qquad \begin{array}{r} 2\,0\,7 \\ -\,1\,6\,9 \\ \hline 1\,\, \end{array}\ \times$$

The derivation of this bug assumes that the student hasn't learned how to borrow yet. When the student attempts to take a larger number from a smaller one, an impasse occurs. The repair to this impasse is the Noop repair. It causes the column difference action to be skipped. If the procedure is using the boundary between answered and unanswered columns to determine the focus of attention, then after the Noop repair, the procedure will return to focus on the column that it just finished. It won't shift its attention to the next column left, as the bug does. Instead, the procedure will go into an infinite loop examining the same column over and over again. This is clearly a star bug. Not only does the no-data-flow hypothesis prevent the generation of an observed bug, it causes a star bug to be generated.

Moving beyond the subtraction skill, one finds that there are procedures that clearly need some kind of "current focus pointer" in order to traverse lists without the aid of visual markers. For example, children can add long columns of digits. This seems to require some kind of register or a counter or something that indexes down the digits in a column. To represent the traversal with a pure push down automaton, one would have to have distinct states for each digit in order to have distinct patterns to fetch that digit. If the push-down automaton were finite, then there would be some finite limit on the number of digits the student could add. This seems totally unlike human mathematical skill.

There is another argument against the position that procedures do not maintain some kind of current focus pointer. It is a *reductio ad absurdum* argument. Consider taking a subtraction test. Under the no-data-flow hypothesis, the patterns in the procedure are used to distinguish the column being worked on from the others by taking advantage of the fact that the columns to the right are answered. However, something must also specify the *subtraction problem* being worked on. To do so, the procedure's patterns might use the fact that the exercise problem is the one that has only solved exercises before it and unsolved exercises after it. Going one step further, the patterns must specify which piece of paper is the test paper. Clearly, the patterns are being burdened with quite a bit of description. The no-data-flow hypothesis entails that patterns mention things that are irrelevant to subtraction. It makes silly predictions. It might predict that a student would believe that a subtraction problem can only be done on a chalkboard or in a textbook, since that is the only place the student sees examples being done. It seems that there has to be some current focus pointer somewhere in order to have the procedure retain any degree of modularity at all.

11.2 Focus is not globally bound

The previous section showed that the procedure is somehow storing and maintaining a current focus of attention. This section compares two ways to do this: globally bound variables (registers) and locally bound variables. In the interest of factoring the hypotheses of the theory as independently as possible, it will *not* be assumed that the control structure is recursive. This makes the nomenclature more awkward, but gives the resulting conclusions a little more generality. One other assumption is needed before the main argument can be presented. It will be assumed that the Backup repair exists. The defense of this assumption is in section 1 of appendix 9. Some important features of this repair are most easily described with an example of its operation.

Figure 11-1 gives an idealized protocol. It illustrates a moderately common bug (Smaller-From-Larger-Instead-of-Borrow-From-Zero). In the Southbay sample of 375 students with bugs, five students had this bug. The (idealized) subject of figure 11-1 does not know all of the subtraction procedure. In particular, he does not know about borrowing from zero. When he tackles the problem 305–167, he begins by invoking a Sub1Col goal. Since 5 is less than 7, he invokes a Borrow subgoal (episode *a* in the figure), and immediately the first of borrowing's two subgoals, namely Borrowing-from (episode *b*). At this point, he gets stuck since the digit to be borrowed from is a zero, which cannot be decremented in the natural number system. He is at an impasse. Several repairs can be used at impasses to get unstuck. The one that interests us here is called the Backup repair. It gets past the decrement-zero impasse by "backing up," in the problem solving sense, to the last goal which has some open alternatives. In this case, there are five active goals:

a.
$$\begin{array}{r} 305 \\ -\,167 \\ \hline \end{array}$$
In the units column, I can't take 7 from 5, so I'll have to borrow.

b.
$$\begin{array}{r} 305 \\ -\,167 \\ \hline \end{array}$$
To borrow, I first have to decrement the next column's top digit. But I can't take 1 from 0!

c.
$$\begin{array}{r} 305 \\ -\,167 \\ \hline 2 \end{array}$$
So I'll go back to doing the units column. I still can't take 7 from 5, so I'll take 5 from 7 instead.

d.
$$\begin{array}{r} \overset{2}{\cancel{3}}{}^{1}05 \\ -\,167 \\ \hline 2 \end{array}$$
In the tens column, I can't take 6 from 0, so I'll have to borrow. I decrement 3 to 2 and add 10 to 0. That's no problem.

e.
$$\begin{array}{r} \overset{2}{\cancel{3}}{}^{1}05 \\ -\,167 \\ \hline 142 \end{array}$$
Six from 10 is 4. That finishes the tens. The hundreds is easy, there's no need to borrow, and 1 from 2 is 1.

Figure 11–1
Pseudo-protocol of a student performing the bug Smaller-From-Larger-Instead-of-Borrow-From-Zero.

1. Borrow-from: a goal that normally just decrements a digit
2. Borrow: a goal that processes columns that require borrowing
3. Sub1Col: the main column processing goal
4. Multi: a goal that traverses across multiple columns
5. Sub: the top-level goal for solving a subtraction problem

The Borrow-from goal has failed. The Borrow goal has no alternatives: one always borrows-from then borrows-into. The next most distant goal, Sub1Col, has alternatives: one alternative for columns that need a borrow, and one for columns that do not need a borrow. Since Sub1Col has open alternatives, Backup returns control to it. The evidence for backing up occurs in episode *c*, where the subject says "So I'll go back to doing the units column." In the units column he hits a second impasse, saying "I still can't take 7 from 5," which he repairs ("so I'll take 5 from 7 instead"). He finishes up the rest of the problem without difficulty.

The crucial feature of the analysis above, for this argument, is that Backup caused a transition from a goal (Borrow-from) located at the top digit in the *tens* column to a goal (Sub1Col) located at the *units* column. Backup caused a shift in the focus of attention from one location to another. Moreover, it happens that the location it shifted back to was the one that the Sub1Col goal was originally instantiated on, even though that column turned out to cause problems in that further processing of it led to a second impasse. So, it seems no accident that Backup shifted the location back to Sub1Col's original site of invocation. Backup shifts both focus and control.

Incidentally, I expect this focus-shifting property of Backup to remain uncontradicted by evidence from other domains. In Newell and Simon's study of eye movements during the solution of cryptarithmetic puzzles, for example, there is ample evidence that backing up (popping a goal in their system) restores not only the goal, but the focus of visual attention that was current when the goal was last active (Newell & Simon, 1972, pp. 323–325).

With the empirical evidence on the table, the basic argument is simple to state: If focus is bound to instantiations of goals, then backing up to a goal automatically restores focus of attention. The schema-instance hypothesis captures the facts quite nicely. The global binding hypothesis runs into trouble. If focus is globally bound (e.g., in a register), then the Backup repair would have to be formulated so that it explicitly resets focus as it sends control back to a goal. But how would Backup know what to reset the focus register to? By hypothesis, the only "memory" for focus is the focus register. Hence, Backup would have to (1) analyze the procedure's structure to figure out how the current focus was calculated, then (2) run these calculations backwards in order to obtain the value that is to be set into the focus register. Clearly, this makes Backup a very powerful repair. Not only can it do static analysis of control structure, but it can simulate a procedure running backwards! It is much more powerful than the other repairs, which do simple things like skipping a stuck action. Backup is so powerful that it can potentially model any conceivable student behavior. This would makes the theory irrefutable, not to mention implausible. In short, if focus is locally bound, Backup is simple; If focus is globally bound, Backup is powerful and implausible.

There are various ways that the global binding hypothesis can be patched up. One can provide multiple focus registers, for instance. As it turns out, there are excellent arguments against such augmentations to the global binding hypothesis. The arguments are rather complex, although quite elegant at times. They have been relegated to an appendix (see section 3 of appendix 9). At any rate, none of the versions of the global binding hypothesis have the empirical and explanatory adequacy that the schema-instance hypothesis has. So the global binding hypothesis will be rejected.

What this means is that representations that do not employ the schemata and instances, such as finite state machines or flow charts with registers, can be dropped from consideration. This puts us, roughly speaking, on the familiar ground of "modern" representation languages for procedures, such as stack-based languages, certain varieties of production systems, certain message passing languages, and so on.

11.3 The applicative hypothesis

The deletion operator (chapter 7), regardless of how it is formalized exactly, is a valuable tool for examining data flow. Having an operator that mutates the knowledge representation allows one to infer the structure of the representation. An important use of this tool is to uncover one of the tacit constraints on data flow.

A prominent fact about bugs is that none of them require deletion of focus shifting functions. For example, if one knows about borrowing-from, one knows to borrow from a column to the left. No bug has been observed that forgets to move over before borrowing-from. This fact deserves explanation.

In all the illustrations so far, focus shifting has been embedded in rule actions (right hand sides). This is no accident. Suppose one did not embed them, but made them separate actions, as in

Goal: Borrow (C) Type: AND
1. (Borrow-into C)
2. (C ← (Next-column C))
3. (Borrow-from C)

The "←" represents a variable setting operation (i.e., a SETQ). A star bug is generated by deleting rule 2. This star bug would borrow from the column that originates the borrow:

*Borrow-From-Self:

$$\begin{array}{r} 3\ 4\ 5 \\ -\ 1\ 0\ 2 \\ \hline 2\ 4\ 3 \end{array}\ \checkmark \qquad \begin{array}{r} 3\ 4^{1}\overset{14}{5} \\ -\ 1\ 2\ 9 \\ \hline 2\ 2\ 5 \end{array}\ \times \qquad \begin{array}{r} 2^{1}\overset{9}{0}{}^{1}\overset{16}{7} \\ -\ 1\ 6\ 9 \\ \hline 1\ 3\ 7 \end{array}\ \times$$

In order to avoid such star bugs, focus shifting functions must be embedded. A constraint upon the knowledge representation is needed. About the strongest constraint one can impose is to stipulate that the language be *applicative.* That is, data flows by binding variables rather than by assignment. There are no side effects: a goal cannot change the values of another goal's variables, *nor even it's own variables.* The only way that information can flow "sideways" is by making observable changes to the external state, that is, by writing on the exercise problem.

The applicative hypothesis is extremely strong, forcing data to flow only vertically. The procedure can pass information down from goal to subgoal through binding the subgoal's arguments. Information flows upward from subgoal to goal by returning results. No counterexamples to the applicative hypothesis have been found so far.

Applicative data flow enables context-free subprocedure acquisition

The applicative hypothesis has a profound effect on learning. It makes learning subprocedures context-free. That is, learning a hierarchical procedure becomes roughly equivalent to inducing a context-free grammar. The basic idea is that the applicative hypothesis, together with the recursive control hypothesis (chapter 9), force data flow and control flow to exactly parallel each other. To put it in terms of grammars, the data flow *subcategorizes* the goals. This, in turn, makes it much easier to induce the goal hierarchy from examples.

Inducing a procedure's calling hierarchy (i.e., goal-subgoal hierarchy) from examples has proved to be a tough problem in AI. Neves (1981) used hierarchical examples to get his procedure learner to build hierarchy. However, subtraction teachers do not always use such examples. Badre (1972) recovers hierarchy by assuming examples are accompanied by a written commentary. Each instance of the same goal is assumed to be accompanied by the same verb (e.g., "borrow"). This is a somewhat better approximation to the kind of input that students actually receive, but again it rests on delicate and often violated assumptions. Anzai (1979) uses various kinds of production compounding (chunking) to build hierarchy. However, to account for which of many hierarchies would be learned, Anzai used domain-specific features, such as the pyramids characteristic of subgoal states in the Tower of Hanoi puzzle. The applicative hypothesis cracks the problem by structuring the language in such a way that hierarchy can be learned via a context-free grammar induction algorithm.

11.4 Summary and formalization

The arguments in this chapter have been somewhat complicated although the conclusion reached is a rather simple one. First, it was shown that procedures need to maintain some notion of a current focus of attention. Roughly speaking, the focus of attention is a pointer to a region of the current problem state where some reading or writing actions are going on. To maintain the current focus of attention, some way to store and transmit focus over time is needed. This facility was labelled "data flow." Various ways to construct it were contrasted.

The simplest facility is based on using registers (globally bound variables) as repositories for the current focus of attention. This allows control flow and data flow to be completely independent. However, this independence led to the downfall of this approach. The Backup repair can be assumed to be a minimally simple way to change control, yet empirical evidence shows that whenever it shifts control, it also shifts focus of attention in certain ways. If control flow and data flow are as independent as the register hypothesis has them, then there is no way to explain Backup's tandem shifts of control and focus. If the two flows are independent, why doesn't Backup shift just one and not the other?

The schema-instance hypothesis revises the register hypothesis in a straightforward way by stipulating that focus is somehow stored in close association with the instantiations of goals. Hence, whenever a goal is resumed, as by the Backup repair, then its stored focus of attention becomes current. In a sense, this hypothesis is a direct response to the difficulties of the register hypothesis. It stipulates that whenever control pops, focus of attention is restored too.

The applicative hypothesis goes one step further. It stipulates the converse: whenever control does not pop (or push), then focus of attention does not change. That is, the only way to change focus is to push or pop goal instantiations. There is no way to change an instantiation's stored focus once that instantiation has been made. The currently executing instantiation can't even change its own focus. This extremely strong hypothesis is motivated by an apparent lack of certain kinds of bugs. If focus could be changed without changing control, then it ought to be possible for students to forget to do such a change. That is, the rule describing the change could be deleted. Yet no such deletions have been found. Indeed, when such deletions are carried out, they result in star bugs. Hence, to explain the way deletion appears to work, data flow must be applicative. This conclusion is captured in the following hypothesis:

> *Applicative data flow*
> Data flow is applicative. The data flow (focus of attention) of a procedure changes if and only if the control flow also changes. When control resumes an instantiation of goal, the focus of attention that was current when the goal was instantiated becomes the current focus of attention.

The impact of this hypothesis on the emerging representation of procedures is fairly simple. Goals are equipped with *arguments.* An argument is a local variable that can be used in the rules that define the goal's subgoals. When a rule calls a goal, it provides values for each of the arguments of the goal that it is calling. For instance, in the goal Borrow:

Goal: Borrow (C) Type: AND
1. (Borrow-into C)
2. (Borrow-from (Next-column C))
3. (Diff C)

the argument of Borrow is C. When rule 2 calls the goal Borrow-from, it provides a binding for Borrow-from's argument by evaluating the focus shifting function Next-column. (The next chapter shows that functions are not a good way to represent the shifting of focus; patterns are better.) The other rules simply pass the current focus of attention, held in C, to the goals that they call.

Passing intension versus passing extensions

Computer science has invented several ways to pass arguments. The most common is call-by-value. Others are call-by-name, call-for-result, call-by-reference and lazy evaluation. The basic dimension of variation is how much of the "meaning" is passed along with the value or datum. From a logician's viewpoint, the issue is whether the objects being passed via arguments are *intensions* or *extensions.* The most common convention is to pass extensions — call-by-value. Lazy evaluation is perhaps the closest approximation in computer programming to passing intensions.

The intension-extension dimension is a valid issue for the theory to examine. Certainly the theory has to take a stand on it if its model is going to be implemented on a computer. The issue is essentially whether focus of attention is a specific geographic region in the problem state or a *description* of a region in the problem state. One might represent an extensional focus of attention by a rectangle in Cartesian coordinates. Intensional focus might be represented by a concatenation of all the patterns that have been used to generate it. Unfortunately, the issue of extensional versus intensional focus is a very difficult one, with some empirical evidence on both sides. The position taken by the theory is to represent focus using parse nodes (see section 2.4), which are halfway between extensions and intensions. Appendix 8 discusses this complex issue.

Chapter 12
Searching the Current Problem State

In the next chapter, the problem of how to represent student's understanding of the problem states will be discussed. It will be shown that students impose a structure on their view of the current problem state. Their understanding is a sort of a task-specific ontology that determines what objects "exist" in the sense that they are relevant to the task. The task-specific ontology also determines the spatial relationships that "exist" between these objects. Regardless of how the students structure their view of the problem state, the students must occasionally search that structure in order to locate information needed during problem solving. This chapter discusses the search issue, which cuts across all the various task-specific ontologies and representations thereof.

The previous chapter showed that procedures maintain a visual focus of attention. It also discussed a certain kind of focus shifting caused by popping goals. However, there are other kinds of focus shifting. For instance, the students write symbols in various locations, presumably shifting focus between each writing action. Such focus shifting requires a procedure-directed movement through the visual-manipulative space. That is, the procedure must search.

The *search problem* is to equip the procedure with facilities that allow it to search the problem state (or rather the student's structured version of the problem state). The essence of the search problem is how much of the search task to represent explicitly in the procedure, and how much to represent below the grain size of the representation as some kind of primitive or underlying facility. That is, the search problem concerns where to place the boundary between the cognitive skill under study, mathematical problem solving, and the perceptual and motor skills that necessarily accompany the exercise of mathematical skill. Three hypotheses will be considered:

1. *Search loops:* The procedure employs explicit search loops in order to access and manipulate the symbols in the problem state.
2. *Path expressions:* The procedure describes a path from the current focus of attention to the desired object. A mechanism that is beneath the grain size boundary actually moves the focus of attention along the path in order to access the desired object.
3. *Pattern matching:* The procedure doesn't need to express anything about *how* to find a desired symbol. Instead, it merely describes *what* it wants. The description is called a pattern. A mechanism called the pattern matcher, which is below the grain size boundary, takes care of actually finding the described symbols.

It will be shown that the evidence is clearly on the side of the third hypothesis.

12.1 Search loops

One way to find symbols is for the procedure to contain search loops. A search loop moves the focus of visual attention across the problem state, stopping when it reaches the location where the procedure will read or write a symbol. For instance, to find the leftmost column, the procedure would loop leftward across symbols until it finds a column (= vertical group of digit symbols) that has lots of blank space to the left of it.

To implement a loop requires at least one disjunction: the conditional that says whether to stop or to go once more around the loop. According to the one-disjunct-per-lesson hypothesis, each such disjunction must be the topic of its own lesson. In itself, this is not bad. It could well be that search loops are taught in a series of lessons.

Consider a simple search that walks down a string of algebraic symbols looking for, say, a variable. This would require a loop (expressed recursively) across the symbols of the string. The loop would be similar in form to the one used in subtraction to walk across the columns of a problem. Given one-disjunct-per-lesson learning, the acquisition of the search loop would mimic acquisition of the multi-column loop:

1. The first lesson concerns the simplest case, where the desired variable is the first element of the string.
2. The second lesson has the variable as the second element of the example string, indicating that it is not the string's initial element but a variable that is being sought.
3. The third lesson closes the loop with examples where the variable is anywhere in the string.

One could imagine a particularly thorough algebra teacher following this curriculum once. However, the representation forces such a three-lesson unit to be presented for each new search! Clearly, students can learn searches without this kind of teaching.

The crucial difference between the multi-column loop and the search loop is exactly that the multi-column loop requires mutation of the problem state at each step and therefore is not really a search loop. To maintain the prediction that the multi-column loop requires several lessons, but the search loop does not, a representational construction is needed that distinguishes the two. The representation needs a special construction to perform searches.

12.2 Pattern matching

In a production system or a Planner-like language, the usual way to access an object is to specify the relations that would be true of it. A typical description might be

```
(AND
  (?X ISA PLACE)
  (?X IN !COL)
  (NOT (?X IS/BLANK)))
```

This description is used to find a non-blank place in the given column. Traditionally, prefixes are used to distinguish search variables such as `?X` from goal arguments, such as `!COL`, which are bound prior to the search. The critical point is that patterns need not specify *how* to conduct the search to locate the described object. They only describe what the search is looking for. The interpreter includes a mechanism called the *pattern matcher* which actually conducts the search.

Essentially, the argument is that since search loops are not taught explicitly in class, some general search mechanism must be in place before instruction begins. Therefore, only the descriptions that drive the search need be learned in class and not the search loops themselves.

Although patterns and pattern matching are the solution that the theory uses for the search problem, it is worth mentioning a special search construction that was once used by Sierra. It possesses many interesting qualities, but turned out rather poorly compared to patterns and pattern matching.

12.3 Function nests as representations of paths

The basic idea of Sierra's old representation was to describe a path between the current focus of attention and the desired new focus of attention. This path was expressed as a nest of functions. For example, if the current focus is a column that requires borrowing, then the Borrow-from goal needs to be called on the top digit of the next column to the left. To shift focus, the following function nest was used:

```
(TopDigit (LeftAdjacentColumn Col))
```

This describes a path. It moves first to the column that is just left of the current focus of attention (represented by the variable `Col`). Then it focuses in on the top digit of that column.

What makes this representation interesting is that it could acquire new descriptive functions in the same way that new subprocedures are acquired. The basic idea is to define functions using the same AND-OR structure that procedures' control structure uses. The crux of the representation was a construction called an *intersection function.* An intersection function is expressed as a functional AND goal. Thus, `LeftAdjacentColumn` is expressed using the following intersection function:

```
Goal: LeftAdjacentColumn (Col) Type: AND, Function?: true
 1. (Columns)
 2. (Left Col)
 3. (Adjacent Col)
```

This function intersects the sets returned by the three subfunctions, `Columns`, `Left` and `Adjacent`. Functions must be set-valued in order to make this work. Thus, a function such as `(Left Col)` returns all places that are to the left of the given column. The result of the intersection function above would be the intersections of all columns, all places to the left of the given column, and all places adjacent to the given column. This means that it returns a singleton set consisting of the left-adjacent column.

Intersection functions can have the same syntax as goals. Hence, learning new intersection functions would be similar, if not identical, to learning new subprocedures. Moreover, rule deletion would be identical to "forgetting" part of a term's definition. The whole concept seems quite tractable. In fact, Sierra used this representation for quite a long time, as representations go. A complete learner was built for it. That is where the fatal flaw was found.

There are too many paths between any two points

By the time the later lessons of subtraction are encountered, a number of intersection functions such as `LeftAdjacentColumn` have been learned. The richness of this set allows long and silly nests of functions to be induced for descriptions. For example, the usual borrow-from place,

```
(TopDigit (LeftAdjacentColumn Col))
```

could also be described by any of the following nests:

```
1. (TopDigit
      (ColumnOfDigit
         (LeftAdjacentDigit
            (BottomDigit Col))))
2. (RightAdjacentDigit
      (TopDigit
         (LeftAdjacentColumn
            (LeftAdjacentColumn Col))))
3. (AboveAdjacentDigit
      (BottomDigit
         (LeftAdjacentColumn Col)))
```

As these nests illustrate, there are many paths to get from one place to another. Path induction will find *all possible paths* from the current focus of attention to the place where attention is next to be focused. When non-cyclic paths are removed, there are still far too many paths. Even when minimal length paths are the only paths induced, there are many paths. Moreover, all the paths are roughly equivalent in that procedures with different paths are observationally indistinguishable.

To avoid this redundancy, one really wants to merge those paths. One wants to describe the *network* instead of all the paths traversing it. That is exactly what patterns do. If a pattern consists of a set of relations among variables, the relations can be viewed as labelled edges for a directed graph with the variables serving as the graph nodes. Thus, patterns express the whole network of relationships between current and successor foci, while a function nest expresses only one path through the network. The reason that path induction generates so many silly, redundant expressions is that it generates all possible paths between two nodes in the network. Clearly, learning is better represented as inducing the network itself. Thus, it need not choose between alternative and nearly equivalent paths.

Paths and relaxation

The discussion above is aimed mostly at establishing a different perspective on patterns rather than criticizing the path framework. The damning problems with paths have to do with the fact that they are hard to "relax." A pattern can be relaxed by deleting one or more of its relations, allowing it to match in more situations than it used to. However, it doesn't work to delete one function from a function nest. Such deletions generally generate nonsensical paths.

Relaxation is used in several ways. It is used by the Refocus repair in order to find a new argument for a stuck action that is "similar" to the argument value that causes the impasse. Relaxation is also used in learning to generalize descriptions in certain situations. Chapter 18 discusses these issues. Suffice it to say that paths are a poor representation for locative descriptions whenever those descriptions must be revised. As long as the descriptions never change, which is not the case here, then path representations work fine.

12.4 Summary and formalization

It is often the case that procedures much search the problem state to find information. If this search were represented explicitly in the procedure, then it would occasionally take the form of a search loop. Such loops would require at least one disjunction in order to terminate the loop when the desired information is found. However, the one-disjunct-per-lesson hypothesis entails that each time such a loop is learned, it would have to be learned in a short sequence of lessons. In many cases, such lessons are not found in today's curricula. Hence, to maintain the truth of one-disjunct-per-lesson learning, a facility for doing search that is beneath the grain-size boundary must be added to the representation. A hypothesis to capture this conclusions is:

> *Pattern*
> Procedures have patterns which are matched against the current problem state.

There are many issues introduced by the addition of patterns to the representation. Chapter 13 discusses what the set of relations should be for patterns and how the student's understanding of the problem state, which is what patterns match against, should be represented. Chapter 14 discusses the expressive power that patterns should be given. They must have at least relations, such as `(LeftOf x y)` or `(Column x)` in order to describe the kinds of information that search seeks. However, it is so far an open question whether they have logical constructions such as quantifiers, disjunctions and so on. Chapter 17 shows that procedures need two kinds of patterns. *Test patterns* are used for the applicability conditions of rules. *Fetch patterns* are used for the focus shifting that occurs when a rule calls a subgoal. Chapters 18 and 19 discuss how patterns are acquired.

Chapter 13
Ontology

This chapter discusses how to represent a student's understanding of the current problem state, or rather, that portion of the current problem state that the student considers relevant to the problem solving task. It is assumed that the student knows or believes that only certain kinds of objects and relations are relevant to learning and problem solving in the given task. That is, it is assumed that the student has a *task-specific ontology* which says what kinds of objects and relations exist in a given problem state when that state is viewed in a task-oriented way. This ontology is specific to the task the procedure solves because the kinds of objects and relations that are relevant varies with the task. The relevant objects for subtraction are different than those for algebra, for tic tac toe, or for drawing cartoons, even though all these tasks are carried out on paper. Indeed, the task-specific ontology may even vary across subjects performing the same task. This variability is the essential problem. It will be shown that the choice of objects and relations used in an inductive learning model has a direct effect on the output of the learner. But the ontology is task-specific, so the theorist must provide it, at least in part. The theorist can control the predictions of the model by controlling the objects and relations used to represent the student's task-specific ontology. If the theorist is allowed total freedom in choosing the model's objects and relations, then the empirical adequacy of the theory may depend more on the cleverness of the theorist than on the principles of the theory. The theory may have little explanatory value.

To summarize, there are two horns to the dilemma: Since the student's task-specific ontology varies across tasks (and perhaps across students as well), the theory must leave its formal expression as an open parameter in the model. Some tailoring must be allowed. On the other horn, if the ontology parameter is too unconstrained, the theory may be vacuous. The problem is to provide some way to constrain the representation of task-specific ontologies. This chapter discusses three solutions to that problem:

1. *Problem state spaces:* The theory places no constraints on the representation of task-specific ontologies. For each task (or possibly each student), the theorist provides a problem state data structure, some functions and relations for accessing it, and some state change operators.

2. *Aggregate object definitions:* Under this approach, the theory asserts that all students have the same conception of two-dimensional space for all mathematical symbol manipulation tasks. That spatial conception is based on a few fundamental concepts, including adjacency, sequence and the compass points: horizontal, vertical and the two diagonals. The variation in ontologies across tasks and individuals is confined to aggregate objects. That is, different tasks will group the symbols differently. Subtraction cares about columns but algebra doesn't. Different students might group symbols differently. Some algebra students group "2+3x" such that "2+3" is an aggregate object. Although grouping strategies may vary, the set of basic spatial relations and the set of state change operators are both constant. A subtraction problem is a horizontal sequence of columns; an algebraic expression (e.g., "$-2+3x^2+y$") is a horizontal sequence of signed terms. The model uses exactly the same spatial relations to describe both cases. Essentially, the spatial relations are a fixed, universal set. The objects vary across tasks and individuals. The set of state change operators is also a fixed, universal set. To tailor the ontology parameter of the model, the theorist provides only a set of aggregate object definitions.

3. *Grammars:* This approach takes the same stand on the universal character of spatial relations and state change operators as the object definition approach. However, it expresses that hypothesis differently. The essential difference is that aggregate objects can be defined in terms of other aggregate objects. In particular, an object is defined by a set of grammar rules that may mention other objects. The formalism for grammar rules embeds the fundamental spatial relations: adjacency, sequence and the compass points. The grammar also establishes a part-whole hierarchy of aggregate objects. This hierarchy is lacking from the object definition approach. To tailor the ontology parameter of the model, the theorist provides a grammar.

All three approaches were implemented in Sierra at various times. They are only a small sample of the many ways that ontologies can be represented. More research is needed in this crucial area. For the domain of mathematical symbol manipulation skills, it will be shown that the grammar approach yields the best theory. However, it is not yet clear whether this general approach will work in other domains. I would hesitate to say that a grammar is the right way to represent a nuclear power plant operator's task-specific ontology. Clearly the plant operator's ontology would not be a simple two-dimensional grammar of the kind used in this theory. Perhaps it would be more like the device topologies used in de Kleer's work on causal models of physical devices (de Kleer, 1979; de Kleer & Brown, 1981). This will be a critical issue when the acquisition of procedural skills is studied in other domains than symbol manipulation.

13.1 Problem state spaces

One approach to representing the subject's task-specific ontology is to use a high-level, structural description of the problem state. For instance, the problem state might be formalized as operator-precedence trees for algebra or as matrices for arithmetic. One must also provide a set of descriptive terms for the procedure for use in accessing parts of the problem state data structure or in testing its properties. Examples of such descriptive terms are a function that retrieves the left side of a given equation, or a predicate that is true of two columns when they are adjacent to each other.

This approach is essentially a projection of Newell and Simon's *problem space* approach onto the problem state dimension (Newell & Simon, 1972). Problem spaces contain information that doesn't directly represent the current state of the visible problem. A problem space for chess contains information about the previous moves in the game, for instance. The *problem state space* approach is just a restriction of the problem space approach. It includes only information about the problem state. That is, it represents the subject's internal representation of the current external state of the problem.

A fundamental tenet of the problem space approach used by Newell and Simon is that problem spaces vary across individuals and tasks. The problem state space therefore is left as a model parameter that can be tailored by the theorist to fit individual subjects. This is its Achilles heel. For several years while repair theory was being developed, Sierra's solver used problem state spaces. The solver's performance was relatively insensitive to variations in the problem state space. However, when the learner was developed, it showed enormous sensitivity to the problem state space. Many important details of the student procedures induced by the learner were controlled solely by the problem state space. Since the learner was basically an inducer, slight shifts in how problems were described were lifted by generalization into the acquired procedure. Since the problem state space has to be fitted by the theorist to the data, the theorist can tailor the learner's output to be just about anything. This gives the theory a great deal of tailorability. Indeed, it can be argued that it gives it too much.

The generation of Always-Borrow-Left depends on the problem state space

The problem state space has enough tailorability that it even affects the generation of a premier bug of induction: Always-Borrow-Left. This subtraction bug is a particularly clear example of how induction can generate bugs as well as correct procedures. Its generation assumes that the student has learned only part of the lesson sequence for subtraction. In particular, it assumes that the student has just been introduced to borrowing. In all textbooks that I know of, the lesson that introduces borrowing uses only two column problems, such as *a*:

$$\text{a.}\quad \begin{array}{ccc} & 5 & \\ & 6 & {}^{1}5 \\ - & 1 & 9 \\ \hline & 4 & 6 \end{array} \qquad \text{b.}\quad \begin{array}{cccc} & & 5 & \\ & 3 & 6 & {}^{1}5 \\ - & 1 & 0 & 9 \\ \hline & 2 & 5 & 6 \end{array} \qquad \text{c.}\quad \begin{array}{cccc} & 2 & & \\ & 3 & 6 & {}^{1}5 \\ - & 1 & 0 & 9 \\ \hline & 1 & 6 & 6 \end{array}$$

Multicolumn problems, such as *b*, are not used. Consequently, the student has insufficient information for unambiguously inducing where to place borrow's decrement. The correct placement is in the left-adjacent column, as in *b*. However, two column problems are also consistent with decrementing in the leftmost column, as in *c*. Given only two column examples, induction can't discriminate between the two placements. The bug Always-Borrow-Left results from the learner taking the leftmost generalization, rather than the left-adjacent generalization, which is the correct one. Always-Borrow-Left produces the kind of solutions shown in *c*. The bug occurred six times in the Southbay sample of 375 students with bugs. Its existence is prime evidence that induction plays an important role in procedure acquisition.

The problem state space has total control over whether or not Always-Borrow-Left is generated. If the problem state space does not include "leftmost column" as one of the descriptive terms, then Always-Borrow-Left is not induced. To cover the data, the problem state space has to have "leftmost column" in it. Yet if the problem state space has all plausible descriptive terms in it, then induction will generate star bugs. For instance, if "tens column" is a descriptive term in the problem state space, then two-column worked examples will generate a star bug that could be called *Always-Borrow-From-Tens-Column. Its work is shown in *d* and *e*:

$$\text{d.}\quad \begin{array}{cccc} & & 14 & \\ & 6 & {}^{1}5 & 6 \\ - & 1 & 9 & 0 \\ \hline & 5 & 5 & 6 \end{array} \qquad \text{e.}\quad \begin{array}{ccccc} & & & 3 & \\ & 6 & {}^{1}5 & 4 & 1 \\ - & 1 & 9 & 0 & 0 \\ \hline & 5 & 6 & 4 & 1 \end{array}$$

The star bug decrements the top digit in the tens column even when that column has already been answered, as in *e*, or is in the process of getting answered, as in *d*. The model should not generate *Always-Borrow-From-Tens-Column. Therefore, the problem state space should not include the descriptive term "tens column."

These examples demonstrate that the empirical adequacy of the the theory is highly sensitive to the problem state space. If the theorist tailors the problem state space, then many bugs and star bugs are not explained, they are merely represented by the presence or absence of descriptive terms. Adjusting the problem state space to include "leftmost column" and exclude "tens column" doesn't explain why one and not the other is a salient descriptor to students. Tailorability reduces the explanatory value of the theory.

Problem state spaces reduce the forecasting ability of the theory

There is a second reason that a tailorable problem state space decreases theoretical adequacy. When the theory is applied to a new task domain, it has to be given a new problem state space. Suppose that data have not been collected for this task domain. With nothing to tailor the problem state space to, the theorist must rely on intuition in formulating the problem state space. Since there is no reason to believe in the theorist's guess for a problem state space, there is no reason to believe the theory's predictions, which depend directly on the problem state space. The predictions merely reflect the theorist's intuitions. Hence, the theory is useless for applications that wish to use it instead of extensive data collection projects. As an example of such an application, suppose someone had just invented some paper-and-pencil tools for calculation and a curriculum to teach people how to use them (e.g., a new way to solve fraction addition problems). This theory would be nearly useless for assessing the quality of that curriculum given that the problem state space had to be guessed. Tailorability reduces the forecasting ability of the theory.

13.2 Aggregate object definitions: fixed spatial relations

During the period that Sierra used problem state spaces, the set of notational terms was adjusted in order to maximize the empirical adequacy of the learner's predictions. The resulting set of primitives had several regularities. For instance, there were several clusters of primitives that expressed the idea of a sequence of objects. A horizontal sequence of columns was represented by a cluster of primitives consisting of:

leftmost column
left adjacent column
column A is left of column B
rightmost column
right adjacent column

There were several of these sequence clusters. It seemed that a powerful underlying concept, sequence, was not being captured by the representation in its most general form. The next hypothesis, the aggregate object definition hypothesis, aims to rectify this.

The basic idea behind the aggregate object definition approach is to split off general spatial notions from task-specific notions. The concepts that vary across tasks and subjects are mostly concerned with aggregation of symbols into groups. This part of the problem state space has to be left open for the the theorist to adjust. Everything else can be fixed. In particular, spatial relations are represented by the following set of predicates on objects:

Topological	
`(Adjacent? x y)`	x is adjacent to y
`(Inside? x y)`	x is inside y
Sequence	
`(Last? S x)`	x is the last element of sequence S
`(First? S x)`	x is the first element of sequence S
`(Middle? S x)`	x is neither the first nor last element of S
`(Ordered? S x y)`	x is before y in the sequence S
Compass points	
`(LeftOf? x y)`	x is to the left of y
`(Above? x y)`	x is above y
`(Superscript? x y)`	x is diagonally up and right of y
`(Subscript? x y)`	x is diagonally down and right of y

There are conventions for ordering sequences. The first element of a horizontal sequence is the leftmost element, and the first element of a vertical sequence is the top element.

The spatial relations are a fixed set, provided by the model. To complete the representation of a student's task-specific ontology, the theorist provides a set of type and part relations for each aggregate object and its parts. For instance, the theorist could define an algebraic equation as an aggregate object by defining one type relation and three part relations:

`(Equation? Q)`	true if `Q` is an equation
`(Lhs x Q)`	`x` is the expression on the left of the equation `Q`
`(Sign x Q)`	`x` is the sign (usually =) between the equation's halves
`(Lhs x Q)`	`x` is the expression on the left of the equation `Q`

As argued in chapter 12, patterns are used for all interface operations between the procedure and the problem state space. In particular, they are used as the applicability conditions for rules and for shifting the focus of attention when a rule calls a goal. Patterns contain both spatial relations and the relations provided by the theorist to define aggregate objects. The following pattern might be used to test if an algebraic equation-solving problem has been completed:

```
(Equation? LQ)
(Equation? Q)
(Last? S LQ)
(Middle? S Q)
(Above? Q LQ)
(Lhs x LQ)
(Variable? x)
(Rhs y LQ)
(Expression? y)
```

This pattern describes two equations, `LQ` and `Q`. They are vertically aligned in some sequence `S`, and `LQ` is last in the sequence. The left side of `LQ` is a variable, and the right side is an algebraic expression. This pattern would match *b* but not *a:*

```
a.   2x+5 = 9                    b.   2x+5 = 9
     2x   = 9-5                       2x   = 9-5
                                      2x   = 4
                                      x = 4/2
```

Both problem states have a vertical sequence of equations, but the pattern doesn't match problem state *a* since its vertical sequence of equations does not end with an equation that has a single variable as the left hand side.

The point of fixing the spatial relations is to remove a degree of freedom from the representation of task-specific ontologies. The only freedom left is the aggregate object definitions. In a sense, the theory has been augmented with a micro-theory of two-dimensional space. This micro-theory increases the explanatory value of the theory as a whole. For instance, one of the principles of the micro-theory is that whenever there is a sequence, the first and last elements of the sequence are salient to the student. If the inducer sees a problem state where `x` is the first element of the sequence `S`, then `(First? x S)` will always be induced as a part of the description of `x`. In particular, if `x` is the leftmost column in a sequence of columns, the induced description of `x` will include the constraint that it is the leftmost column. This explains why Always-Borrow-Left is a bug. The column it borrows from is a leftmost column. Induction is forced to predict the bug given a training set of two-column borrow examples. The micro-theory explains the bug from the general principle that people always notice the boundary points of linear arrangements of objects.

13.3 Grammars

With the addition of a micro-theory of space and a symbol-level grain size, the tailorability of the model is drastically reduced. The way that the model is adapted to represent varying conceptions of notational syntax is limited to defining new kinds of aggregate objects and their associated part relations. The set of primitive spatial relations and primitive writing operations remains the same.

This section discusses several hypotheses about the *meaning* of pattern relations based on this representation of task-specific ontologies. In particular, what are the runtime implications of the spatial relations and the relations that define aggregate objects? From the perspective of building a computer model, the issue is how to define the meaning of the relations that are used in patterns. If the pattern has `(Column x)` in it, how does the pattern matcher enforce this constraint on the bindings of the pattern variable `x`? That is, what is the relationship between pattern relations and the current problem state?

The myopia problem

The most straightforward relationship between patterns and problem states is simply to provide each term with some geometric definition. For example, the spatial predicate, `(Adjacent? x y)` might be defined to mean that there is nothing but blank paper between x and y. By this definition, the 3 and the x are adjacent in both *a* and *b*:

a. 3 x = 6

b. $\frac{3}{4} = \frac{x}{8}$

Not only are the 3 and the x separated only by blank space, but they are the same distance apart. By any local definition of `Adjacent?`, the 3 and the x are adjacent in both *a* and *b*. This strange interpretation of *b* is not one that subjects make.

This myopic behavior was discovered in an early version of Sierra. At first I thought it was an instance of the old "how near is near" problem that has plagued AI for at least a decade (Denofsky, 1976). How close do two symbols have to be to be adjacent? To my knowledge, no one has solved the "how near is near" problem, if indeed "solving" it makes any sense in the abstract. There are a collection of hacks for getting around it, mainly involving fudge factors and manipulations of the grain size of the coordinate system. However, as various increasingly desperate hacks were applied to fix Sierra's myopia, it became clear that the approach of using local geometric definitions for terms was just too local to be workable.

The robustness problem

Another approach is to maintain locality of a sort by using definitions that search for maximal conditions within a neighborhood. For instance, `(Adjacent? x y)` could be defined as "there is no object which is closer to x than y, and closer to y than x." In other words, x and y are the closest objects to each other. This would correctly rule out adjacency for **3x** in

$$\frac{3}{4} = \frac{x}{8}$$

since the the closest object to the 3 is the equals sign, not the x. These sort of locally maximal definitions are an improvement over absolute definitions, but even they have problems.

For one thing, they require special cases for symbols that aren't roughly circular in shape, as the digits and letters are. A fraction bar is one such symbol. One wants 3 and x to be adjacent in the following fraction:

$$\frac{3\ \ x}{2}$$

However, the bar is the closest symbol to the 3, not the x. Bars have to be made an exception to the rule in order to get the 3 and the x to be adjacent.

A second problem is a lack of robustness. A little sloppiness in the placement of symbols changes the truth values of predicates defined with locally maximal definitions. For instance, in the first line of

$$\begin{array}{llllll} 3 & & x & & = & 6 \\ & y & & = & 2 & \end{array}$$

The 3 and the x are not adjacent because the y on the second line is too close.

Adjacency is the foundation of symbol groups in mathematical notation. It plays the role of string adjacency in text parsing or temporal adjacency in speech understanding. If adjacency can't be well defined, then the chances of an adequate definitions for other notational terms is poor indeed.

The local ambiguity problem

There is a second kind of problem with local definition of relations. It involves local ambiguity. A local ambiguity occurs when there are several interpretations for a certain subset of the problem state, yet all but one interpretation fail to fit into an interpretation of the whole problem state. That is, there is ambiguity when only a part of the scene is considered, but the ambiguity disappears when the whole scene is considered. Take, as an example, the string "2+3x." One interpretation is that "2+3" is an expression. While locally correct, this interpretation cannot be extended to include the x (assuming a correct syntax for algebraic notions). To see why filtering local ambiguities is important, suppose the procedure wants to extend the expression by appending "+5" to its right end. It uses some pattern to fetch the current expression. If the page bears "2+3x" it is possible that the pattern matcher will return "2+3" as the expression. Appending "+5" to this will cause x to be overwritten. This is not a mistake that people make. Clearly, the definitions of pattern matching must be modified so that such local ambiguities will not be returned.

One solution to the local ambiguity problem is to have each term's definition check the context of its group of symbols as part of determining whether it is true of them. For instance, an algebraic expression with two terms could be defined as:

```
(BinaryExpr x y) ≡
   (Term? x) ∧
   (Term? y) ∧
   (Adjacent? x y) ∧
   (LeftOf? x y) ∧
   [∀z (= z y) ∨ ~(Adjacent? z x) ∨ ~(LeftOf? z x)] ∧
   [∀z (= z x) ∨ ~(Adjacent? y z) ∨ ~(LeftOf? y z)]
```

The intent is for this relation to match "2+3x" but not "2+3." This definition uses universal quantifiers to check that there is nothing just left or just right of the expression. Thus, it rules out "2+3" because it sees the "x" just to the left of it.

Actually, the definition is a little too strong. It must allow certain symbols at its sides, such as equal signs, for instance. So each disjunction would need to be extended with literals such as

```
...∨ (Equal (Read z) '=) ∨ (Equal (Read z) '|)
```

In fact, whenever a new notational symbol is learned, all definitions, such as this one, for aggregate objects that can be located adjacent to the new symbol must have their definitions updated. For instance, when "≠" is learned, then the definition above would have to be extended. This would make it difficult, perhaps, to formulate a plausible theory for incremental learning of notation.

Actually, using context-sensitive definitions would probably not work in general. Consider the string "2+3|5−8|". The extended definition for `BinaryExpr` given earlier allows "|" on its side. So it correctly calls "5−8" an expression. However, it also calls "2+3" an expression. In order to discriminate between the two, it needs to look further than one symbol away.

Whether or not this approach of using look-ahead in local definitions of terms will solve the local ambiguity problem in general amounts to asking whether every mathematical notation has an LR(k) grammar (or equivalently, whether it is a deterministic language). It has been shown that all precedence languages are deterministic languages (Floyd, 1963). The class of precedence languages takes in the sort of linear mathematical expressions used in computer languages. However, it is uncertain whether *two-dimensional* mathematical notation is a precedence language or even a deterministic language. In short, there is reason to doubt whether the local definition approach will always be sufficiently powerful to express all mathematical notation. Of course, learning definitions with many symbols of look-ahead may grossly complicate a theory of their acquisition.

Using grammars to solve problems with myopia and local ambiguity

When relations are defined using only local geometric knowledge, relations such as "term" and "expression" have no relationship to each other except perhaps for co-occurrence in some patterns in the procedure. Consequently, there is no way to prune objects which satisfy their local definition but fail to participate in a global parse of the image. If notational objects are defined by a grammar, the definitions of objects refer to other objects by name. This provides information linking the objects together. It can be used to solve problems of myopia and local ambiguity during pattern matching.

The basic idea is that relations can have fairly sloppy, individual definitions if the definitions are used in concert. A relation in a pattern matches a form in the problem state only if that form participates in a *global parse* of the problem state. The grammar as a whole acts as a filter on possible instantiations of the relations. Hence, the local parse of "2+3" as an algebraic expression in "2+3x" is ruled out because it does not fit into a global parse of the whole problem state. Thus, the grammar is used to filter out local ambiguities. Similarly, grammars solve the myopia problem of pattern matching. In

```
3   x   =   6
  y   =   2
```

the 3 and the y are not adjacent because there is no global interpretation of the problem state that groups those two symbols together as a unit. A local definition of `Adjacent?` might suggest that they are adjacent, but the global interpretation would filter this suggestion out before the pattern matcher can mistakenly retrieve it.

A technical detail: maximal parses

To insist that the problem state have a complete parse is a little too strong. The current problem state may not have a complete parse. This is often the case in the midst of problem solving. En route from "2+3x" to "2+3x+5", the problem state is "3+3x+", which is not a well-formed algebraic expression. Since it hasn't a complete parse, pattern matching would not be permitted to access *any* part of it. So the complete-parse restriction is a tad too strong.

One stipulation that works is to specify that objects must participate in a *maximal* parse. A parse is maximal if the group it covers is not a proper subset of any other parse's cover. Since "2+3" is a proper subset of "2+3x", it is not a maximal parse of the problem state "2+3x+". Hence it is not accessible to matching. There are stipulations other than this one that work, but they seem to produce exactly the same filtering as the maximal parse stipulation. At any rate, some kind of global coherency is necessary as a filter on matching, although currently the details of what that coherency is don't seem to be too important.

13.4 Summary and formal hypotheses

The arguments in this chapter concern how to restrict the ontology parameter of the model. The model uses a set of relations and state change operators to represent the way students structure their views of the current problem state. In most cognitive modeling efforts (e.g., Newell & Simon, 1972), this parameter is left tailorable. Depending on the task and the subject's perception of it, the theorist constructs a different representation of the problem state, along with the pattern relations that are used to access it. This approach, dubbed the problem state space approach, gives a great deal of tailorability to the model.

It was argued that the degree of tailorability was too high. The argument turned on the fact that the present theory is an inductive learning theory. (N.B., Newell and Simon's theory of human problem solving (1972) is not a learning theory, so its predictions may be less sensitive to the tailoring of the problem space.) Subtle changes in the relations that described the problem state or the operators that changed it are lifted up by induction and placed into the procedure. Hence, the theorist may control the output of the learner by tailoring the problem state space. This reduces the theory's ability to explain why some procedures are acquired and others are not.

The ontology parameter cannot be completely fixed since subjects' perceptions of the task really are different across tasks and individuals. However, certain parts of their understanding do not change much, given that the domain is limited to mathematical symbol manipulation tasks. These less variable kinds of knowledge can be fixed by the theory. The basic idea is that the subject's notions of two-dimensional space do not change much across tasks or from one individual to another. However, the way that symbols are grouped into aggregate objects does vary. The theory can fix the spatial relations, but it needs to leave the specification of aggregate objects open to tailoring.

Two ways to capture this basic idea were discussed. One is to allow the theorist to define aggregate object relations, using Lisp or some other convenient language. The relations act locally, recognizing instances of themselves in the problem state. For instance, `(Column x)` might be true of vertically adjacent pairs of digits. This way of defining relations turned out to have severe problems. Notions such as adjacency don't really depend much on the local geometric relations between two symbols, but rather on the problem state as a whole. This *gestalt* aspect of spatial knowledge forces the representation to use the definitions of objects and spatial relations in concert to filter out interpretations that people would not make of the problem state.

To make this cooperative filtering possible, objects are defined by a grammar. The grammar uses the fundamental spatial relations as part of definitional formalism. This way of defining aggregate objects is expressed by the following three hypotheses:

> *Spatial relations*
> The following 5 relations are the spatial relations:
>
> | | |
> |---|---|
> | `(First? S x)` | Object x is the first part of some sequential object S. |
> | `(Last? S x)` | Object x is the last part of some sequential object S. |
> | `(Ordered? S x y)` | Object x comes before y in some sequential object S. |
> | `(Adjacent? S x y)` | Object x is adjacent to y in some sequential object S. |
> | `(!Part x y)` | Object x is a part of object y. |
>
> *Grammars*
> Aggregation of symbols into groups is defined by a spatial grammar based on the notions of sequence, part-whole and the compass points: horizontal, vertical and the two diagonals. For each aggregate object defined by the grammar, a new categorical relation is defined.
>
> *Relations*
> The relations available to patterns are the spatial relations, the categorical relations defined by the grammar, and the usual arithmetic predicates.

The relationship between the grammar and the pattern relations is implemented by categorical relations. When a new object is defined by the grammar, a new categorical relation becomes available for used by the patterns. For instance, the grammar might define a multidigit number with the following two rules:

```
NUM ---> DIGIT
NUM ---> DIGIT  (DIGIT)+  DIGIT     ; HORIZONTAL
```

(The formalism for grammar rules has not been motivated yet. It is discussed in section 15.2. It is based on some ordinary context free grammar conventions: parentheses mean a category is optional, and + means a category may be repeated arbitrarily many times.) The first rule says that a number can be just a single digit. The second rule says that a number can be two or more digits in a horizontal sequence. Whenever a rule has more than one category on the right side, it must be annotated with one of the compass points: horizontal, vertical, superscript or subscript.

The grammar's definitions cause pattern relations to become defined. In this illustration, the rules cause a categorical relation, `NUM`, to be defined for patterns. `(NUM x)` is true of the form `"23."` Multi-category rules establish part-whole relations. When `"23"` is parsed by the second rule above, `(!Part x y)` is true when x is the 2 and y is the `NUM` object, 23. Similarly, the other spatial relations depend on the grammar for their meaning. `(Ordered? S x y)` is true when S is the number, x is the 2, and y is the 3.

The categorical relation **NUM** is not true of the 2 alone in the form 23. One of the main purposes of using a grammar to represent aggregate objects is to filter out such local parses. This was argued for in section 13.3. It is captured by the following hypothesis.

> *Global matching*
> The set of objects in a problem state that patterns can match against is limited to those that participate in a maximal parse of the problem state as determined by the grammar.

This hypothesis is responsible for keeping "2+3" from being treated as an expression when the whole problem state is "2+3x." It can be implemented many ways. Sierra implements it by parsing the problem state bottom-up using the grammar. This yields a set of parse trees. Each parse tree covers some set of symbols in the problem state. The parse trees that do not cover a maximal set of symbols are deleted. Those that remain contain, as parse nodes, all the possible objects that patterns may match against.

Representing state-changes

The preceding discussion centered on how students view a static, unchanging scene: the current problem state. It was assumed that students have an ontology that says what kinds of aggregate objects and relations are relevant to the task at hand. The task-specific ontology structures how students view a single problem state. By symmetry, there ought to be a similar task-specific knowledge source that structures their view of *changes* in the problem state. I think that there is such knowledge, but I admit to being quite confused on the subject. At the crux of state changes lies the notorious frame problem of AI: how does one handle the fact that only a little bit of a problem state changes at time, so almost all references into it may remain unchanged. A central concern is whether foci of attention should refer extensionally or intensionally. These difficult issues are discussed in appendix 8.

Chapter 14
Pattern Power

Chapter 12 showed that the interface between procedures and problem states should use patterns and pattern matching. However, this leaves many issues unresolved. An important unresolved issue concerns how much descriptive power patterns may have. This issue basically asks what kinds of logical constructions are used in patterns, or equivalently, what the representation language for patterns is.

14.1 Viewing patterns as logics

A convenient and natural way to discuss the power of pattern languages is to equate them with logics. A pattern corresponds to a proposition. What the pattern matches against corresponds to a model, in the logician's sense of the word "model." Matching a pattern is equivalent to satisfying the corresponding proposition in the model. Although this is a standard way to look at patterns, an example might be helpful to bring it into sharper focus. A typical pattern from a production system or Planner-like language is:

```
{(?X ISA PLACE) (?X IN !COL) (NOT (?X IS/BLANK))}
```

Pattern variables are indicated by a "?" prefix; variables that are bound before the pattern is matched are indicated with a "!" prefix. What this pattern means is "give me a place `X` that's inside the given `COL` and not blank. The equivalent in a first-order logic would be

$$\exists x\ (\mathtt{Place}\ x) \wedge (\mathtt{In}\ x\ \mathtt{COL}) \wedge \sim(\mathtt{Blank}\ x)$$

The quantifier is existentially bound because the pattern should fail to match only if there are no blank places in `COL`. It should fail in a null (empty) model, for example. If x were universally bound, the proposition would be true in the null model.

The order of the logic

There are a number of constraints illustrated by the example. First, the logic must be at least first order. A propositional (variable-less) logic hasn't the expressive power needed to mention several notational objects at once. Both x and `COL` have to be mentioned in the preceding example. The need to mention several objects at once is entailed by the need to shift focus in the procedure. Since focus was shown to be necessary for mathematical procedures (section 11.1), the pattern logic must be at least first order.

There are many higher-order logics. Since they include first-order logic, parsimony counsels considering them only if first-order logics prove to have inadequate expressive power. So far, the expressive power of first-order logics has been sufficient to allow formulation of an empirically adequate theory.

Clausal form

Having established the order of the pattern logic, one can ask whether all the descriptive power of first-order logic is necessary. A convenient way to do this is to examine, in turn, each logical connective, quantifier and syntactic device. This approach is complicated by the redundancy of first-order logic. Almost any expression using a given construction can be converted to a logically equivalent expression that does not use the construction. To convert an examination of connectives and other devices into an examination of descriptive power, we need to eliminate this redundancy.

An easy way to eliminate redundancy is to use a normal (or canonical) form. The normal form that makes this discussion clearest is *clausal form* (see any textbook on mathematical logic, e.g., Yasuhara, 1971). Any proposition in a standard first-order logic can be converted to clausal form in four steps:

1. Remove implications; e.g.,
   $\sim\forall x\exists y\ (P\ x\ y) \wedge [\forall z\ (R\ x\ y\ z) \supset (Q\ y\ z)]$
   becomes $\sim\forall x\exists y\ (P\ x\ y) \wedge [\forall z\ \sim(R\ x\ y\ z) \vee (Q\ y\ z)]$

2. Push negations down to literals; e.g.,
   $\sim\forall x\exists y\ (P\ x\ y) \wedge [\forall z\ \sim(R\ x\ y\ z) \vee (Q\ y\ z)]$
   becomes $\exists x\forall y\ \sim(P\ x\ y) \vee [\exists z\ (R\ x\ y\ z) \wedge \sim(Q\ y\ z)]$

3. Skolemize. That is, convert existentially bound variables into Skolem (anonymous) functions. The arguments of the Skolem function are any universally bound variables whose scope includes the existential quantifier. Nullary Skolem functions are expressed as Skolem (anonymous) constants; e.g.,
   $\exists x\forall y\ \sim(P\ x\ y) \vee [\exists z\ (R\ x\ y\ z) \wedge \sim(Q\ y\ z)]$
   becomes $\sim(P\ a\ y) \vee [(R\ a\ y\ (f\ y)) \wedge \sim(Q\ y\ (f\ y))]$
   where *a* is a Skolem constant and *f* is a Skolem function.

4. Convert to product-of-sums form, that is, a conjunction of literals or disjunctions; e.g.,
   $\sim(P\ a\ y) \vee [(R\ a\ y\ (f\ y)) \wedge \sim(Q\ y\ (f\ y))]$ becomes
   $[\sim(P\ a\ y) \vee (R\ a\ y\ (f\ y))] \wedge [\sim(P\ a\ y) \vee \sim(Q\ y\ (f\ y))]$

While this version of clausal form is not quite a normal form (order within disjunctions and conjunctions has not been stipulated), it yields a short check list of constructions:

1. predicates
2. conjunctions
3. Skolem constants (wide scope existential quantifiers)

4. constants
5. functions
6. negation

7. variables (universal quantifiers)
8. Skolem functions (narrow scope existential quantifiers)
9. disjunctions

This list is the topic of the chapter. The discussion centers on which of these expressive facilities the pattern language should have. It will be shown that the first three are necessary, the next three are optional, and the last three should be prohibited.

14.2 Predicates, conjunctions and Skolem constants

The first three constructs are fundamental to patterns. All three are needed to express even the simple pattern mentioned earlier. The pattern and its equivalent clausal form expression are:

```
{(?X ISA PLACE) (?X IN !COL) (NOT (?X IS/BLANK))}
```

```
(Place a) ∧ (In a COL) ∧ (NonBlank a)
```

The predicates and conjunctions are obvious. The pattern variable, ?X, has been converted to a Skolem constant, *a*. It will be assumed that predicates, conjunctions and Skolem constants are a part of the pattern representation language.

14.3 Constants, negations and functions

In the Planner-style pattern,

```
{(?X ISA PLACE) (?X IN !COL) (NOT (?X IS/BLANK))}
```

the variable `!COL` is bound outside the pattern. When the pattern is converted to clausal form,

```
(Place a) ∧ (In a COL) ∧ (NonBlank a)
```

it is converted to a quasi-constant, `COL`. `COL` behaves like a constant with respect to pattern matching in that the matcher doesn't try to assign a binding to it. It behaves like a variable in that its value (i.e., the binding assigned to it outside the pattern), not its name, is what is used by the pattern matcher. In order to interface patterns with the data flow machinery that manipulates focus of attention, `COL`-like quasi-constants are needed in patterns.

For regular constants, such as numbers, it is a moot point whether they are in the pattern language. The expressive power of numeric constants can be had by adding primitive arithmetic relations to the language. Thus, to eliminate `(Equal x '5)`, one employs `(Five x)`. So far, no empirical consequences have been discovered that could discriminate patterns with constants from patterns without them. As it turns out, Sierra's implementation of the grammar automatically generates such "constant" relations, naming them with the symbols themselves: `(5 x)` means `(Five x)` and `(+ x)` means that `x` is a plus sign. These are used instead of constants in patterns.

Functions are similar to constants. By manipulation of the set of predicates, one can eliminate these devices. To eliminate functions, one uses relations: (P y (f x)) becomes (P y w) ∧ (F w x). This variability can be used to express the show-work principle as a constraint on the syntax of patterns. The issue is discussed in detail in section 15.1. However, some basic ideas will be briefly presented here. The show-work principle says that if a subprocedure is to be learned, any intermediate results of its computations must be written down in on the page or chalkboard in the worked exercises that teach it. This has implications for the use of arithmetic functions because they produce "invisible objects," namely numbers that are not usually present in the problem state. One use of patterns is in the applicability conditions of rules. They are used to test whether a rule may be run by the interpreter in the current problem state. If arithmetic functions are used there, then the value that the arithmetic function produces is never written down. It is calculated as part of matching the pattern, but it is never passed further. This use of functions can't be learned when learning obeys the show-work principle. Hence, arithmetic functions can be omitted from patterns that serve as applicability conditions.

Like constants and functions, negation can also be eliminated by modifying the set of pattern relations. In clausal form, negations only appear on literals. Consequently, negation can be eliminated by doubling the set of pattern relations: the negation of each pattern relation is added if it doesn't already exist. To eliminate `(Not (> T B))`, one uses either `(Not> T B)` or `(≤ T B)`. So the issue of whether to permit negations and functions in the pattern logic is moot with respect to expressiveness. The convention that Sierra uses is to allow negations on arithmetic relations only. The set of spatial and categorical relations (see section 13.4) is designed in such a way that negation is not needed for them.

14.4 Disjunctions are needed for variables and Skolem functions

Disjunctions, variables and Skolem functions are closely related. Clearly, if the logic forbids variables (universal quantifiers) then there will be no Skolem functions (existentials inside the scope of universal quantifiers). Somewhat less obvious is that forbidding disjunctions guts variables of their expressive power.

In practice, most universally quantified expressions have the form `∀x(P⊃R)`. The `P` expression defines a domain of quantification to be some subset of the objects of the problem state. The `R` expression asserts something about the objects in that subset. Consider, for example, an expression that might be useful in algebra:

```
∃x∃y (LikeTermsP x y) ∧
     [∀z (Between? x z y) ⊃ ~(LikeTermsP x z)]
```

This asserts that `x` and `y` are like terms, and that everything between `x` and `y` is not a like term to them. This expression might be used to find the closest term `y` that can be combined with `x`. The universally quantified sub-expression has the typical form `∀x(P⊃R)`. It says that none of the objects between `x` and `y` are like terms to `x`. If `x` were the first term of

$$2p^2 + 3r + 5p^2 + 6p^2$$

then `y` would be matched to $5p^2$ instead of $6p^2$. When the expression is represented in clausal form, it becomes

```
(LikeTermsP a b) ∧
[~(Between? a z b) ∨ ~(LikeTermsP a z)]
```

where *a* and *b* are Skolem constants, and *z* is a Skolem variable. The point is that the implication has become a disjunction. If the clausal form forbids disjunction, then the usual `∀x(P⊃R)` expressions cannot be used. The only ones that can be used have a pure conjunction or a single literal as the interior. Such expressions would assert something of *every* object in the problem state. This is rather useless for pattern matching since it doesn't discriminate among various objects. It says something about the problem state as a whole, but not about how to tell the desired objects from the others. It is difficult to believe that variables (universal quantifiers) would ever be used in patterns if they could not employ disjunction. In short, if there is no disjunction, then there's no need for variables (universal quantifiers) and hence no need for Skolem functions (narrow scope existential quantifiers).

14.5 Disjunctions in patterns

Chapter 4 showed that inductive learning of disjunctions is impossible unless some constraint is placed on the occurrence of disjunctions in generalizations. It was shown that the observed procedures could be induced if induction was constrained either to choose the minimal number of disjunctions or to learn at most one disjunction per lesson. The latter position was shown to explain the almost universal use of lessons as an instructional aid. As such, it was preferred in the theory on grounds of explanatory adequacy. This, and other arguments, motivated the acceptance of the one-disjunct-per-lesson hypothesis. The hypothesis applies to patterns as well as control structure, of course. Disjunctions in patterns are not introduced during ordinary inductive learning of a subprocedure. Instead, each disjunction is the subject of a lesson itself.

This assertion has an immediate entailment: disjunctive notational concepts must be represented in such a way that they can be used in any pattern. To see this, suppose that "column" is a disjunctive notational concept. This is not so implausible since there are two kinds of subtraction columns:

```
  6 5
-   3
-----
```

The tens column has one digit; the units column has two. Suppose further that "column" is not represented in a way that allows it to be shared among patterns. To describe "leftmost column" requires the notion of "column," which is disjunctive, so "leftmost column" is disjunctive. Since learning a disjunctive pattern requires a lesson of its own, "leftmost column" would require a special lesson. To describe "left-adjacent column" would require another lesson. Every pattern employing the notion of "column" would have a disjunction, and one-disjunct-per-lesson entails that each such disjunction must have its own lesson. Clearly, this is not how notational knowledge is acquired. Instead, the concept "column" is taught once as a notational term. Afterwards, any pattern that employs the notion of columns just uses the term's name, e.g., as a predicate `(COLUMN x)`. So, disjunctive notational concepts must be represented in a way that allows them to be shared among patterns.

To put it differently, learning notation involves learning the definitions of terms: once a term like "column" is defined, a token standing for it may occur in any pattern. Disjunctions therefore occur only in the definitions of notational terms, and not in patterns. But what knowledge base has these term definitions? Clearly, this argument has provided independent motivation for the *grammar:* it is a repository for definitions of notational terms. Earlier, in section 13.3, we inferred its existence as a solution to the myopia, robustness and local ambiguity problems of pattern matching. Here its existence has been supported as an entailment of one-disjunct-per-lesson. This convergence is a weak, but gratifying argument in support of banning disjunctions from patterns. Although it is logically void (because it is an abduction not a deduction), it seems to indicate that we are on the right track. To reiterate the basic idea, a pattern can't say "its either two vertically aligned digits *or* a digit over a blank." It can only say "its a column" and the grammar defines "column" with the disjunctive description "a column is two vertically aligned digits *or* a digit over a blank."

This application of one-disjunct-per-lesson makes empirical predictions. As an example, consider the predicate `LikeTermsP`. This predicate could be defined as

```
(LikeTermP x y) ≡
   (Term x) ∧
   (Term y) ∧
   [∀w (Factor w x) ⊃
    [(Number w) ∨ ∃z (Factor z y) ∧ (IsomorphicP z w)]] ∧
   [∀w (Factor w y) ⊃
    [(Number w) ∨ ∃z (Factor z x) ∧ (IsomorphicP z w)]]
```

This stipulates that `x` and `y` be terms that have identical factors, except for numerical factors. Hence, $3x^2y$ is a like term to $2y3x^2$ but not to $3x^2$. `LikeTermsP` must be defined using universal quantifiers and disjunctions. There is no way to express it without at least one disjunction. Consequently, if a student doesn't know the definition of `LikeTermsP` before being shown how to combine terms, it is predicted that the student won't induce the correct patterns for that transformation. To do so, the student would have to induce disjunctions in patterns, and that is ruled out by the one-disjunct-per-lesson learning principle.

Remarkably, every algebra text that I have examined has a short lesson teaching `LikeTermsP` before the first lesson on combining like terms. This supports the prediction that non-primitive predicates with disjunctive definitions are taught in their own lesson.

Another way to learn aggregate objects

Although many notational objects in algebra are introduced with explicit lessons, this is not generally the case for arithmetic notational objects. In particular, the notational term "column" is not introduced in its own lesson. Instead, it appears that "column" is taught by using a special device. In section 15.2, it is shown that lines, such as the bar used in subtraction problems, do not obey the same grammatical conventions as other symbols. Instead, they are apparently used to mark the boundaries of forms. (This idea was suggested to me by Jim Greeno.) Thus, the bar of subtraction marks the boundary between the answer row and the rest of the problem. Carrying this idea one step further, lines might be used to *teach* new notational concepts. When subtraction problems are first introduced, all the textbooks that I have seen use lines to mark columns. Examples are

	tens	units
	3	7
-	1	5
	2	2

	tens	units
	3	7
-		5
	3	2

The vertical and horizontal lines indicate how to parse the problem state. In particular, they indicate that there are two columns. The columns are even named. Given enough drawings like this and the convention that lines mark boundaries, the learner can learn the aggregate object term "column" without an explicit lesson devoted to the subject. Although not much is known yet about how new notational terms are acquired for the student's grammar, it seems that the basic position of applying one-disjunct-per-lesson learning to grammar acquisition is quite plausible.

14.6 Summary and formal hypotheses

The issue discussed in this chapter is how much expressive power to give to patterns. It is shown that the only really critical question is whether disjunctions are allowed in patterns. If they are banned, then universal quantification and narrow scope existential quantification are no longer useful, so they can be dropped.

Whether to have disjunctions in patterns is a tricky issue. It is clear that disjunction cannot be completely omitted from the interface because some notational concepts must employ it. However, inductive learning of disjunctive concepts is an impossible task unless induction is strongly biased or constrained in some other way. The proposed solution is twofold: notational disjunctions are learned with special devices, such as an explicit lesson, that tells the inductive learner how to formulate the disjunction. This is a simple application of the one-disjunct-per-lesson hypothesis. The second half of the solution is to note that it only makes sense to acquire a disjunctive concept as part of the definition of a new notational term (aggregate object). Doing so makes the concept available for other patterns, and not just the pattern at hand.

Putting these two halves together implies that disjunctions, and by implication, universal quantifications as well, occur only in the definitions of notational terms. Notational terms are defined in the grammar. Although the descriptions in a grammar may be complex, patterns are simple. In particular, since patterns lack disjunction and universal quantification, they are reduced to simple conjunctions of relations. By suppressing the logical connective $\wedge$, a pattern can be represented even more simply as a set of pattern relations. The arguments of the pattern relations are either pattern variables (i.e., Skolem constants) or goal arguments. A pattern relation can also be inclosed in a negation. That is all the logical machinery that is needed. Patterns can be just that simple. These considerations are captured in the following hypothesis:

> *Conjunctive patterns*
> Disjunctions, universal quantifiers and narrow-scope existential quantifiers are banned from patterns. Semantically, a pattern is a conjunction of possibly negated predicates on existentially quantified variables, functions and constants.

This hypothesis allows functions and constants in patterns since it was shown that their inclusion or exclusion is basically a syntactic matter. The issue will be dealt with (briefly) in the next chapter.

Chapter 15
Syntax of the Representation Languages

Almost all the major aspects of the knowledge representation language have been discussed. Only one major feature remains to be discussed. Chapter 17 will show that test patterns (applicability conditions) should be distinct from fetch patterns (focus shifting patterns). This distinction will be assumed herein so that this chapter may complete the discussion of the representation language by defining its syntax. That is, the syntax of goals, rules and grammar rules will be fixed. Even at this seemingly inconsequential level of detail, there are a few problems whose solutions impact the theory's predictions. For instance, it makes a difference whether grammars are represented as first-order logics or as context-free grammars. However, the impact of such competing alternatives is minor compared to the kinds of representation issues that have already been discussed. Most of this chapter will simply describe the choices taken by the theory; there will be little discussion of the alternatives. Some readers may wish to skip this chapter.

15.1 Syntax of the procedure representation language

Chapter 10 argued that goals have a binary type to distinguish AND goals from OR goals. There are three traditional syntaxes for binary-typed goal structures: CFGs (context-free grammars), ATNs (Augmented Transition Nets), and AOGs (And-Or Graphs). The issue here is essentially a topological one. There is no difference in the expressive power of the representations. Trivial algorithms exist to translate an expression in one (i.e., any AOG, any CFG, or any ATN) into an equivalent expression in the other. Where the three representations differ is their effect on operations that manipulate them as structures. Their shape affects how elegantly and parsimoniously each operation can be formalized. Clearly, it doesn't effect whether or not the operation *can* be formalized. If the operation can be defined for any of them, one can translate expressions in the others into the tractable representation, perform the operation, then translate the result back into the original representation. So it is only the elegance of the theory that is at stake here.

CFGs are not compatible with trivial ORs

In a CFG, OR goals are represented by non-terminals and AND goals are represented by the right sides of rules (see figure 15–1a). In an ATN, AND goals are represented by levels, and OR goals are represented by states (see figure 15–1b). Both CFGs and ATNs naturally generate "trivial" goals. A trivial goal has just one subgoal. CFGs generate trivial AND goals corresponding to rules with just one category on the right side (e.g., A → B). ATNs have trivial OR goals whenever a state has just one arc leaving it (e.g., the first state of the ATN of figure 15–1b). Trivial ORs are a convenience in subprocedure acquisition. To acquire a new subprocedure in an ATN, one simply adds a new arc. For instance, figure 15–1c shows the ATN of 15–1b with a new subprocedure, E, added to it. Because of the implicit trivial OR at the first state, the new subprocedure could be added without making any structural changes to the old ATN. When the representation uses trivial OR goals, the assimilation conjecture (section 10.1) can be interpreted in a quite literal fashion: acquiring a new subprocedure changes *none* of the old goal structure.

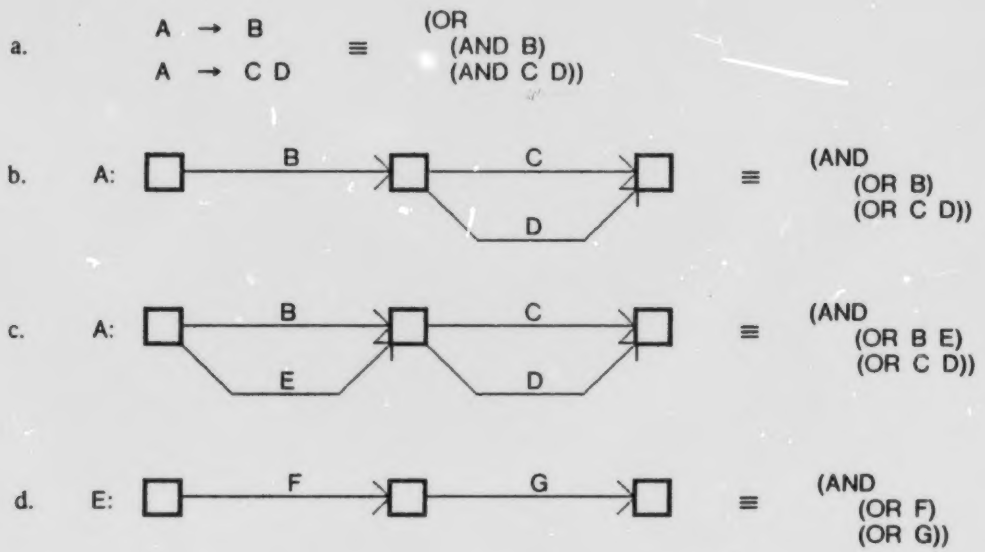


Figure 15–1
A CFG and several ATNs, with their logical equivalents.

Trivial ORs are worth having in order to allow subprocedure assimilation to be simple and elegant. To get them requires a little extra work. Whenever a new subprocedure's AND has more than one subgoal, the new subgoals are placed inside trivial ORs. If the new subprocedure E of figure 15–1c had subgoals, as in figure 15–1d, they would be placed in trivial ORs. This means that all the disjunctions that the learner could possibly use are already in the goal structure; to add a subprocedure, the learner just adds a new disjunct to some existing disjunctions. Automatic addition of trivial ORs is natural in the ATN syntax. It can be stipulated for the AOG framework. For CFGs, stipulation won't work. Adding the extra rules that are needed for the trivial ORs also introduces trivial ANDs. These trivial ANDs clutter up the goal structure, making the Backup repair and other structure-sensitive operations go awry. So the choice of three syntaxes is narrowed to two: ATNs and AOGs

ATNs will not let AND rules shift focus

In an AOG, both AND goals and OR goals have arguments. In an ATN, only the AND goals have arguments. They are called *registers*. Each ATN level (= AND goal) has its own locally bound registers. The applicative hypothesis entails that focus shifting occur only in the actions of the OR rules (arcs) since these are what call ATN levels. In an AOG, both AND rules and OR rules may shift focus. It will be shown that it is better to use focus shifting on AND rules only. Since this is just the opposite of the ATN convention, it entails that AOGs are a better syntax than ATN.

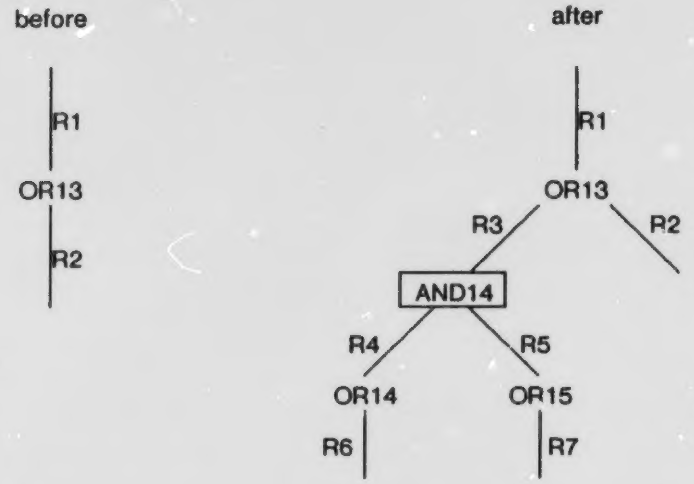


Figure 15–2
An AOG fragment before and after a new subprocedure's acquisition.

In the ATN syntax, when a new subprocedure is acquired, a new arc and (in general) a new level are added to the ATN. The new level needs to be given registers. The adjoining arc needs to be given a focus shifting function. For instance, suppose the learner acquired the ATN of figures 15–1c and 15–1d given the ATN of figure 15–1b. The learner would have to provide registers for level E and a focus shifting function for the arc of 15–1c that calls it. In addition, the learner has to induce the focus shifting functions for each of the actions of the new level, namely arcs F and G. The simplest way to provide a focus shifting function for the adjoining arc (arc E) is to make it a "null focus shift," that is, simply pass the registers of the calling level (level A) down to the called level (level E). This entails that all levels would have exactly the same register contents — focus would never be shifted, except just before a primitive action. This simple way of deciding the adjoining arc's focus will not allow even a correct subprocedure to be acquired. It's unworkable.

Another simple tactic is to assign to the adjoining arc the focus shifting function that would have been given to the first arc of the new level (arc F in figure 15–1d). That is, the focus that is appropriate for the first action is made the focus of the entire level. Although the details won't be presented here, this tactic will not work either. For instance, it won't allow the main column loop of subtraction to be acquired.

Two simple methods have failed to assign the adjoining arc a focus shifting function. Some complicated method appears necessary. However, a simpler path is to abandon ATNs and let only the AND rules bear the focus shifting function. OR rules will just pass the focus of the caller to the callee. Figure 15–2 illustrates the acquisition of a new subprocedure under these conventions. The adjoining rule, R3, receives no focus shifting function. It just passes OR13's arguments to the new AND, AND14. On the other hand, rules R4 and R5 are assigned focus shifting functions. Under the ATN syntax, R6, and R7 would receive the focus shifting functions that R4 and R5 receive in the AOG syntax, but the ATN syntax also requires that R3 have a focus shifting function. The problematic focus shift of R3 is avoided if the AOG syntax is used.

No applicability conditions on AND rules

It has been shown that subprocedure adjunction is simplest if (1) the procedure includes trivial ORs, and (2) only AND rules have focus shifting functions. To these conventions, one can add the observation that AND rules have no use for applicability conditions. All rules of an AND goal will be executed in order. If it is learned that the first subgoal of `(AND A B)` is optional, the structural modification will be:

```
(AND (OR A) (OR B)) → (AND (OR A Noop) (OR B))
```

where Noop is an action that makes no change in the problem state. The point is that trivial ORs mean that all knowledge about applicability can be captured on OR rules. There is no variability in the sequence or applicability of the AND rules. Hence, applicability conditions can be omitted on AND rules.

Facts functions on OR rules only

For some actions, facts functions are required. For instance, when simple borrowing is first acquired, the borrow-from action is to (1) fetch the next column's top digit, and (2) subtract one from it. The focus shift (1) must be on an AND rule, Borrow's first rule in fact. The issue is whether to put the facts function (2) in the same place. If it is on the AND rule, then both the focus and the decremented number will be passed to the trivial OR that is between Borrow's first rule and the action that writes the new number down. When BFZ (i.e., borrow from zero) is acquired, it will be adjoined beneath this trivial OR. It will be passed the ORs arguments to use as its arguments. With respect to the focus portion of the trivial ORs arguments, this makes good sense. However, it makes little sense to pass the decremented value of the top digit. In fact, that value won't even be defined since a zero would have to be decremented to obtain it. Clearly the facts function `Sub1` must be *beneath* the trivial OR if BFZ is to be acquired. That is, Borrow and its trivial OR must have the following syntax:

```
BORROW (COL) Type: AND
  1. <a fetch pattern that binds T to the top digit of the next column to the left of COL>
     ⇒ (OR13 T)
  2. ...

OR13 (TD) Type: AND
  1. true ⇒ (OverWrite TD (Sub1 (Read TD)))
```

The fetch is on the AND's rule 1, but the facts function is on the OR rule.

This example prompts the general constraint that whenever a new subprocedure has an action involving a facts function, the function nest is separated from the fetch pattern and placed on the rule of the trivial OR corresponding to the action.

This convention makes it simple to state the show-work principle: all invisible object descriptions are represented by functions, such as the facts functions. These are located on a special place on OR rules, namely the argument positions of the subgoals. With this convention, the show-work principle amounts to (1) prohibiting functions in patterns and (2) limiting functions nests to containing at most one function that produces an invisible object (i.e., `(Sub1 (Read T))` is okay but `(Sub1 (Sub1 (Read T)))` is not). This means that pattern syntax is very simple: it is a conjunction of relations whose arguments are all variables and goal arguments. Relations such as

`(Equal? X (Sub1 Y))` are banned. This syntactic convention entails that relations that might be most rationally written with a `Read` function, e.g., `(LessThan? (Read T) (Read B))`, are better written directly in terms of variables as is `(LessThan? T B)`. With this definition of the facts predicates, the prohibition against functions in patterns can be made total.

Summary

The conclusion is that the procedure representation language should be an AOG language subject to the following restrictions:

> *Trivial ORs:* Every AND goal and every primitive goal is a subgoal of some OR, even if that OR has only one subgoal.
>
> *OR rules don't fetch:* OR rules do not have fetch patterns. Their only use for patterns is as applicability conditions — determining whether or not to run.
>
> *AND rules don't test:* AND rules do not have applicability conditions. Their patterns are used for shifting the focus of attention.
>
> *AND rules don't have facts functions:* Functions which create invisible objects are contained in the actions of OR rules only.
>
> *Patterns have no functions:* Neither test patterns nor fetch patterns have functions. A pattern is represented as a set of relations on variables.

These conventions mean that AOGs can use a simple syntax for rules: a rule has 4 parts:

1. The name of the goal it's under (the goal).
2. The name of the goal it calls (the action's subgoal).
3. A list of functions and/or variables that provide arguments for the subgoal (the action's arguments).
4. A pattern. This is interpreted as a fetch pattern for AND rules and a test pattern (applicability condition) for OR rules.

Patterns are simply sets of relations. However, for convenience, the non-grammatical facts predicates (e.g., `LessThan?`) may be negated. Spatial and categorical relations are designed in such a way that negation is not needed for them.

15.2 Syntax of the grammar representation language

The student's notational knowledge is assumed to be a context-free grammar. Although it was implied in chapter 13 that the grammar should have the descriptive power of first-order logics, CFGs do not have quite that much power. In particular, CFGs have difficulty representing notational terms that would be easily represented if the representational language had universal quantifiers. So far, I know of only one such term: `LikeTermsP` (see 14.4). It cannot be represented in a CFG, or at least in the CFG language that Sierra uses. Nonetheless, CFGs have many computational advantages that outweigh this minor lack of expressiveness.

Expressing the universal spatial relations

A main purpose of the grammar representation language is to embed the spatial relations that are held to be task- and subject-independent. There are four spatial ideas:

Part-whole	Aggregate objects have parts.
Adjacency	One object is next to another and no object is between them.
Sequence	Aggregate objects are sometimes sequences, in which case the spatial relations First, Last, Before and After are well defined among the elements of the sequence.
Direction	For mathematics, the compass points — horizontal, vertical, superscript and subscript — are the important directions.

In general, the simplest CFG languages employ only part-whole and adjacency. The rule A → B C means that B and C are parts of A and that they are adjacent. Whenever such a rule occurs in the grammar, the categorical relation (A x) is defined. Moreover, when the rule is used to parse some objects, call them *a*, *b* and *c*, then the following relations are automatically true:

`(!Part a b)`	Object b is a part of a.
`(!Part a c)`	Object c is a part of object a.
`(Adjacent? a b c)`	Object b is adjacent to object c.
`(Adjacent? a c b)`	Object c is adjacent to object b.

To add the idea of sequence to the grammar language is easy. The standard artifice of a Kleene plus is used, but in a restricted context. If an aggregate objects is a sequence, it is defined by a rule whose right side has three elements:

```
EXPR  ⇒  TERM  (SIGNED-TERM)+  SIGNED-TERM
```

This rule means that an algebra expression (`EXPR`) is a list whose first element is a term. The last element is a signed term (i.e., a term with + or − ahead of it). It may have zero or more intermediate elements. The parentheses mean that the middle element is optional; the Kleene plus means that it can be iterated. The only place a Kleene plus may occur is on the middle element of a three-category right-hand side. This implements the notion of sequence. It also implements the notion that the endpoints of sequences are special. They can be a different category from the interior elements of the sequence. As we see above, the lead element of an algebraic expression may or may not have a sign as its first symbol, but the remaining elements of the expression must have signs.

Whenever some symbols are parsed by a rule such as the one above, certain relations automatically are true of them. For instance, if *a* is parsed as an `EXPR` with *b* and *c* as its parts, then the following relations would be true:

`(First? a b)`	Object b is the first element of sequence **a**.
`(Last? a c)`	Object c is the last element of sequence **a**.
`(Ordered? a b c)`	Object b comes before c in sequence **a**.
`(!Part a b)`	Object b is a part of a.
`(!Part a c)`	Object c is a part of object a.
`(Adjacent? a b c)`	Object b is adjacent to object c.
`(Adjacent? a c b)`	Object c is adjacent to object b.

That is, the normal part-whole and adjacency relations have been joined by the spatial relations for sequence.

Category redundancy rules

In order to get the parse tree to match the part-whole relations, rules that have just one category on the right, e.g., `A → B`, are treated specially. These rules essentially express a categorical redundancy. `A → B` says that B is an A. Thus, `NUMBER → DIGIT` means that a digit is a kind of number. Such rules are called category redundancy rules. They are parsed differently. If "5" is parsed as a number, then the digit 5 is not a part of the number 5, it *is* the number 5. This convention is needed in order to allow the learner to use the part-whole relations to reduce the complexity of pattern matching and pattern induction. Without it, matching and induction would be much slower, by several orders of magnitude (see section 18.1).

Boxes and the compass points

To represent the geometric information in two-dimensional forms, some definitions are needed. The rectangle that an object fits into is called a *box*. A box has four properties — left, right, top, and bottom — whose values are Cartesian coordinates in the plane of the image. X:Left will mean the location of the left edge of the box of X. Given this nomenclature, the way the grammar represents geometric relationships can be defined. The grammar rules have a modifier that specifies one of the compass points as the direction that their constituents run:

`X ---> Y Z ; HORIZ`	means	Y:Right = Z:Left, and Y:Top = Z:Top, and Y:Bottom = Z:Bottom
`X ---> Y Z ; VERT`	means	Y:Bottom = Z:Top, and Y:Left = Z:Left, and Y:Right = Z:Right
`X ---> Y Z ; SUPERSCRIPT`	means	Y:Right = Z:Left, and either ½(Y:Top + Y:Bottom) = Z:Bottom or Y:Top = ½(Z:Top + Z:Bottom)
`X ---> Y Z ; SUBSCRIPT`	means	Y:Right = Z:Left, and either ½(Y:Top + Y:Bottom) = Z:Top or Y:Bottom = ½(Z:Top + Z:Bottom)

As these definitions indicate, adjacency is defined as two constituents having boxes that share a boundary; they abut. Thus, Y and Z are horizontally adjacent if Y:Right=Z:Left. The relationship of a box to its constituents' boxes is one of containment. In the above rules, the box assigned to X must properly contain those assigned to Y and Z, and furthermore, no other boxes than those of Y and Z may overlap X's box. For example, the parse of a rational equation depends on a gross vertical expansion of the box assigned to the equality sign:

$$\boxed{\frac{3}{4}}\boxed{=}\boxed{\frac{x}{8}}$$

The box for the equal sign contains only "=" and blank space.

In the case of the rules for superscripts and subscripts, a disjunction is needed to handle cases where either the base or the exponent is much larger than the other:

$$\left[\frac{1}{3 + x}\right]^2 \qquad \text{Base:Top} = \tfrac{1}{2}(\text{Exponent:Top} + \text{Exponent:Bottom})$$

$${}_2\left[\frac{1}{3 + x}\right] \qquad \tfrac{1}{2}(\text{Base:Top} + \text{Base:Bottom}) = \text{Exponent :Bottom}$$

Special notational devices: bars, crossouts

Besides the compass points, there are two other rule modifiers. These deal with lines of various kinds, rather than alphanumeric symbols. Mathematics seems to use vertical and horizontal bars not as constituents of objects, but as markers for boundaries. (This idea was originally suggested to me by Jim Greeno.) It solves many problems. For instance the bar of multicolumn arithmetic columns can't be considered a constituent of the problem because it would block one from using columns as constituents. That is, in the problem

$$\begin{array}{r} 687 \\ -230 \\ \hline 257 \end{array}$$

there are three columns, but all of them intersect the same bar. If the bar is treated as a symbol in the same way that the digits are, then it would have to be shared in some way by all three columns. This is impossible in the context-free grammar formalism. The way the current grammar language handles this is to provide a rule modifier that specifies that the boundaries between the constituents paired by the rule be darkened. A rule for subtraction columns would be

```
ACOL    --->  COL  (DIGIT)    ; VERT  BARRED
COL     --->  DIGIT  (DIGIT)  ; VERT  UNBARRED
```

The first rule describes the column as a `COL` above an optional answer digit, separated by a bar. The second rule describes `COL` as a digit above an optional digit, and these must not have a bar between them.

A second rule modifier is needed to handle the scratch marks that students use to cross out symbols. The slash or X put over a symbol is not the same sort of constituent as regular alphanumeric symbols. It overlaps other symbols. No other mathematical symbols overlap. The grammar language provides a special annotation to indicate whether a constituent must be crossed out or not crossed out. For instance, to accommodate the stack of crossed-out numbers that can occur when borrowing across zero, the grammar might use the rules:

```
XNUM    --->  NUM  (/NUM)+  /NUM    ; VERT  UNBARRED
COL     --->  CELL  (%DIGIT)        ; VERT  UNBARRED
CELL    --->  DIGIT
CELL    --->  XNUM
```

The first rule defines an `XNUM` as a number with some crossed-out numbers beneath. The "/" means that the constituent must be crossed out. `/NUM` is a number with a slash or an X through it. A `COL` is defined as either a digit or an `XNUM` on top of an optional digit. The "%" indicates that the digit may not be crossed out.

Chapter 16
Summary: Representation level

The representation level, chapters 9 through 15, discusses what kinds of constraints should be put on the way student knowledge is represented in the model. These hypotheses define a formal knowledge representation language. More importantly, they place constraints on the kinds of learning and problem solving that the model can do. They affect the predictions made by the model. They are chosen to make the model's predictions fit the data. The constraints on representation are empirical hypotheses. Unlike most AI research on representation languages, the aim is not to define a language that allows expression of subtle epistemological distinctions or a language that promotes mental hygiene among the knowledge engineers that use it. The aim is quite different. It is to define a language that is true. The question is, what is it true of? Two answers, the "mentalese" interpretation and the "relevance" interpretation, seem plausible to me.

The mentalese interpretation

One view is that information in the mind has its own structure, the mind's *mentalese* (Fodor, 1975). On this view, the representation language is true of the subjects' mentalese in the same sense that a learning model is true of the subjects' learning. Learning is an internal information process; mentalese is an internal information structure. Neither learning nor mentalese can be directly observed, although their effects can be. The constructions of the representation language, e.g., grammars, goal stacks and the like, are taken as describing mentally held information structures. Hypotheses about the representation language are true or false in the same way that hypotheses in physics are true or false. This interpretation of the hypotheses is simple, traditional and elegant. However, I find it a little hard to square with introspections on my own cognition. The other interpretation, based on relevance, is ontologically verbose but more intuitively acceptable.

The relevance interpretation

A procedure is a way of describing a systematic sequence of actions that change the state of the problem. Defined this way, two issues immediately become apparent. One issue concerns the nature of the interface between the procedure and the environment: What is the vocabulary of manipulative actions that the procedure can employ and what is the vocabulary of descriptions of the environment that the procedure uses in guiding its choices? This issue, the interface issue, asks about the range of primitive, individual input/output actions the procedure employs. The second issue addresses the procedure's internal, runtime state. What kinds of actions can the procedure make to change the internal state? How does the procedure view or structure the internal state?

Presumably, people have much more information than they actually use as they execute their procedures. This holds for both interface information and internal state information. People can see much more on a page that bears an arithmetic problem than they deem relevant to its solution. Similarly, they remember much more about what they have already done in solving it than they they deem relevant. A subject might remember that the last subtraction fact was very hard to remember or that the tens column's borrow was interrupted in order to watch an airplane fly by. The real internal state of a human procedure is just as rich in irrelevant detail as the real written problem.

One interpretation of the hypotheses governing the representation language is that they describe what kinds of information people deem *relevant* to learning and problem solving. On this relevance-based view, when the theory uses non-overlapping aggregated objects, it is not claiming that people can't see overlapping aggregates. When it claims that there is a simple goal stack rather than the spaghetti stack that coroutines use, it is not claiming that people cannot do coroutines. In both cases, the claim is only that when people learn arithmetic, they act as if they believed that only non-coroutine procedures are relevant and only non-overlapping aggregate objects are relevant. The hypotheses are constraints on learning, although they are expressed as constraints on the kind of information structure that are learned.

There is nothing inconsistent with holding both the mentalese and the relevance interpretations of the representation language. It could well be that the structure of mentalese *causes* only certain information to be relevant. It could also be that relevancy *causes* procedural information in the mind to take on a certain structure. The best way to find out what is really going on is to push the empirical examination of the representation language as far as possible. The hope is that when a great deal is known about the kinds of knowledge structures that optimize the fit of various cognitive models to empirical evidence, then the answer to such interpretations will be obvious.

Preview

This chapter summarizes the representation level in two ways. First, it traverses the main principles of the representation, briefly mentioning their supporting arguments. Second, it updates the formal model that was presented in chapter 7, the summary to the architectural level. It will be shown that almost all of the model is entailed by the hypotheses that define the representation. In particular, it is shown that only five issues remain to be discussed in the following level, the bias level.

16.1 The interface issue

It was just mentioned that the central issues of representation are the interface issue and the internal state issue. The interface issue will be summarized first.

The interface issue divides into sub-issues. One concerns the descriptive vocabulary used by the procedure to format its access and manipulations of the problem state. Speaking metaphorically, the issue is how does the procedure *understand* the problem space. In particular, what kinds of objects does it think can exist? What is its private ontology? A common technique used by AI learning models for specifying an ontology over problem states is to equip the procedure with an explicit set of primitive relations. This is not such a good practice since the set must be specified differently for different tasks. A subtraction procedure views its problem states differently than an algebra equation-solving procedure does. Thus, the set must be tailored by the theorists for each task, and possibly for each subject.

A less tailorable alternative is to fix the relations that are relevant in all tasks in the domain and vary only the task-dependent ones. Taking this tack, chapter 13 found that the constant relations were spatial ones: vertical, horizontal, part-whole, adjacency, order, and boundary points. The task-dependent relations all concerned aggregate objects — objects like columns or equations that are groups of other aggregate objects or individual characters. These considerations motivate the following hypotheses:

Spatial relations
The following 5 relations are the spatial relations:

`(First? S x)`	Object x is the first part of some sequential object S.
`(Last? S x)`	Object x is the last part of some sequential object S.
`(Ordered? S x y)`	Object x comes before y in some sequential object S.
`(Adjacent? A x y)`	Object x is adjacent to y in some aggregate object A.
`(!Part x A)`	Object x is a part of aggregate object A.

Grammars
Aggregation of symbols into groups is defined by a spatial grammar based on the notions of sequence, part-whole and the compass points: horizontal, vertical and the two diagonals. For each aggregate object defined by the grammar, a new categorical relation is defined.

Relations
The relations available to patterns are the spatial relations, the categorical relations defined by the grammar, and the usual arithmetic predicates.

The grammar expresses the student's ontology, or rather, that part of the student's ontology that the student considers relevant to the task. The procedure's patterns are couched in terms of the aggregate objects defined by the grammar (via the categorical relations), the spatial relations, and a few arithmetic predicates, such as `LessThan?` and `Equal?`.

The problem state is viewed as a *gestalt*. A problem state usually has many locally well-defined aggregate objects that don't fit into a globally coherent parse. When a procedure manipulates a problem state, it doesn't use those. It uses only the aggregate objects that participate in global parse. A grammar is used in preference to a set of local definitions for aggregate objects because it allows the theory to capture this gestalt use of notational knowledge with a simple hypothesis:

Global matching
The set of objects in a problem state that patterns can match against is limited to those that participate in a maximal parse of the problem state as determined by the grammar.

The grammar serves two purposes: It defines the ontology of the problem space, and it filters out aggregate objects that are incoherent in a gestalt view of the problem.

Given a problem state, the grammar determines the set of objects, aggregate objects and relationships that the procedure "considers" potentially relevant. However, the procedure needs some mechanism to access this field. In particular, it need to *search* this field in order to find appropriate objects to shift its attention to. The search problem is another important interface issue. Chapter 12 argued that search is a skill that is not acquired in the same way that the subtraction procedure is acquired. In particular, there are no lessons that teach search loops. The conclusion is that the search skill is in place before subtraction is taught. This means that procedures need only convey to this preexisting facility what it is that needs to be found. The search facility will find it if it is a part of the problem state. The descriptions are called patterns, and the search skill is called pattern matching. The hypothesis that captures the theory's chosen solution to the search problem is

Pattern
Procedures have patterns which are matched against the current problem state.

Having patterns creates the problem of specifying how much descriptive power they may have. This is a third interface problem — the pattern power problem. Chapter 14 inventories the stock of descriptive devices used in first-order logics. It shows that the one-disjunct-per-lesson hypothesis entails that patterns should not have disjunctions in them. Instead, disjunctive descriptions should be named and inserted into the grammar. Similarly, universal quantifiers should be banished to the grammar along with existential quantifiers when they are used inside the scope of universal quantifiers. This solution to the pattern power problem means the grammar has a new function: it is the repository of disjunctive and universally quantified notational descriptions. The arguments motivated the following hypothesis:

> *Conjunctive patterns*
> Disjunctions, universal quantifiers and narrow-scope existential quantifiers are banned from patterns. Semantically, a pattern is a conjunction of possibly negated predicates on existentially quantified variables, constants and functions.

Arguments in section 15.1 showed that an elegant equivalent to the show-work principle is available: functions and constants were banished from patterns and put instead in a special place on rules. The rule actions may have functions and constants, but patterns are simply relations on pattern variables.

16.2 The internal state issue

The internal state issue concerns how the procedure keeps track of what it is doing, in particular, where it is currently working and what it is intending to do. More accurately, one can divide the internal state into information that refers to regions in the problem state (focus of attention) and information that has no external referent (e.g., goals). These kinds of information correspond roughly to data flow and control flow, respectively. They can be considered to be two halves of the internal state question.

Several arguments were presented in chapter 9 that show that control flow is best modelled as a goal stack. A stack-based, recursive control structure enables the learner to acquire center recursive subprocedures, such as borrowing from zero, without violating the one-disjunct-per-lesson hypothesis. It also allows the Backup repair to be defined as popping the goal stack. Stack popping yields several observed bugs and avoids some star bugs that are generated by other kinds of Backup (e.g., chronological Backup). The arguments motivate the following hypothesis:

> *Recursive control structure*
> Procedures have the power of push down automata in that the representation of procedures permits goals to call themselves recursively, and the interpreter employs a goal stack.

The second issue concerns how the procedure keeps track of its focus of visual attention. Once again, the Backup repair is involved in a crucial argument. It is shown that Backup restores not only the control (goal) component of the execution state, but it restores the focus of attention, as well. This indicates that focus is locally bound. Goals are instantiated with the current focus of attention. When the stack pops to resume a goal (even if it is popped by a repair), the goal's original focus of attention becomes current once again. Moreover, it is shown that once focus is instantiated for a goal, it is shifted only when the goal calls a subgoal. The goal's focus cannot be reset (i.e., there is no SETQ for focus). These two aspects together mean that the procedure is applicative:

> *Applicative data flow*
> Data flow is applicative. The data flow (focus of attention) of a procedure changes if and only if the control flow also changes. When control resumes an instantiation of goal, the focus of attention that was current when the goal was instantiated becomes the current focus of attention.

In short, the procedure moves both kinds of internal state together. Although one, focus of attention, refers to the external world and the other does not, both are stored together on a stack.

Given that the procedure keeps goals on a stack, there needs to be some convention for when to pop the stack. That is, an exit convention is needed to indicate when a goal is satisfied and may be popped. The arguments concerning this issue are a little weak, but there are several and they all point to the same conclusion. An elegant and simple model results when goals are given a binary type. (There is preliminary evidence for a third goal type, a Foreach loop, but it has not yet been incorporated into the model and tested.) AND goals execute all their rules; OR goals execute just one:

> *And-Or*
> Goals bear a binary type. If G is the current goal in runtime state S, then `(ExitGoal? S)` is true if
> 1. G is an AND goal and all its rules have been executed, or
> 2. G is an OR goal and at least one of its rules have been executed.

The central debate over exit conventions concerned the deletion operator. Ultimately, it was shown that the best formulation of the deletion operator was:

> *AND rule deletion*
> `(Delete P)` returns a set of procedures P′ such that each P′ is P with one or more AND rules deleted from the most recently acquired subprocedure.

> *Most recent rule deletion*
> `(Delete P)` returns a set of procedures P′ such that each P′ is P with one or more rules deleted from the most recently acquired subprocedure.

Since this formulation depends crucially on the AND/OR type difference, it supports the And-Or hypothesis.

When the syntax of the procedure is considered (in chapter 15), the And-Or types play a central role. The type of a goal determines not only when a goal is exited but also how its rule's patterns are interpreted. The patterns of an AND goal's rules are interpreted as fetch patterns. They are used to shift the focus of attention. The patterns of OR goals are used as applicability conditions. An applicability condition must be true if the rule is eligible for execution. In the next level, chapters 17 to 20, it is shown that these two kinds of patterns are subject to quite different learning biases. They are quite different not only in function but in content and acquisition. This distinctiveness reflects on the original AND/OR distinction, adding a little more support to the principle.

This completes the synopsis of the hypotheses introduced by the representational level. The remainder of this chapter spins out their implications for the formal model. First the learner is considered, then the interpreter and the local problem solver.

16.3 The learner

At the architectural level, the learner was specified in terms of three undefined functions: `Disjoin`, `Induce` and `Practice`. These functions had to be specified informally since the formalism representations for procedures had not yet been defined. The representation level has specified the needed representation language. The three functions can now be defined. However, it turns out that the constraints imposed by the representation are not quite powerful enough for a complete definition. The learner overgenerates, producing millions of procedures for each lesson. Fortunately, it is not hard to see where the missing constraints go. This subsection explains the definition of the learning model, showing where the missing hypotheses go. The next level, the bias level, discusses what those hypotheses should be. Formally, three new undefined functions will be used to indicate where the missing hypotheses go. The old undefined functions, `Disjoin`, `Induce` and `Practice`, will be given definitions in terms of the new undefined functions. The informal definitions of the old functions that were given in the architectural level are repeated below:

`(Induce P XS)`	represents disjunction-free induction. The first argument, `P`, is a procedure. The second, `XS`, is a set of worked example exercises. `Induce` returns a set of procedures. Each procedure is a generalization of `P` that will solve all the exercises the same way that they are solved in the worked examples. `Induce` is not permitted to introduce disjuncts. If the procedure cannot be generalized to cover the examples, perhaps because a disjunction is needed, then `Induce` returns the null set.
`(Disjoin P XS)`	represents the introduction of a disjunct (e.g., conditional branch) into `P`, the procedure that is its first argument. The second argument, `XS`, is a set of worked example exercises. `Disjoin` returns a set of procedures. Each procedure has had one disjunct introduced into it. The disjunct is chosen in such a way that `Induce` can generalize the procedure to cover all the examples in `XS`.
`(Practice P XS)`	represents another kind of disjunction-free generalization, one driven by solving a set of practice exercises, `XS`. `Practice` returns a set of procedures. Each procedure is a generalization of its input procedure `P`.
`(Delete P)`	represents deletion. Parts of the input procedure `P` are deleted. `Delete` returns a set of procedure resulting from various deletions.

The representation makes a distinction between control structure (goal hierarchy) and data flow structure (goal arguments, rule patterns and rule actions). The simplest way to deal with these two structurally dissimilar kinds of information is to assign their acquisition to different function. `Disjoin` will be in charge of adding the new goal structure; `Induce` and `Practice` will add everything else. That is, `Disjoin` grafts a skeletal version of the new subprocedure onto the old procedure. The skeleton has goals and rules, but the goals lack arguments, the rules lack patterns, and the rule's actions lack arguments. Only the goal topology is fixed by `Disjoin`. `Induce` and `Practice` flesh out the skeletons found by `Disjoin`. They do pattern induction and function induction in order to add patterns and action arguments to the new subprocedure. The reason for dividing the labor this way is that pattern and function induction are disjunction-free inductions. The only disjunct introduced by a lesson is in the goal structure. It is at the parent OR, the place where the new subprocedure adjoins the old procedure. Finding and adding that disjunct is `Disjoin`'s job. It requires a very different kind of algorithm than disjunction-free induction. With these introductory comments said, each of the previously undefined functions will be defined.

Disjoin

In Sierra, `Disjoin` is implemented by a context-free parsing algorithm. Given an example, `Disjoin` parses it using the old procedure as if it were a context-free grammar, and using the example's problem state sequence as if it were a string of primitive actions. It can do this because (1) the procedure representation language is recursive, (2) goals have only two types, AND and OR, and (3) data flow is applicative. These properties are, of course, main results of the representation level. Because they are true of procedures, procedures can be used like context-free grammars to parse examples. When `Disjoin` parses an example, it will not be able to parse it completely using the old procedure. The example uses the new subprocedure, which the old procedure does not have. However, by guessing all possible skeletal subprocedures before it parses, `Disjoin` can figure out which of the possible skeletons will allow the example to be parsed. Let `(Skeletons P X)` be a function that returns all skeletons that allow `P` to parse the example `X`. The next step is to apply `Skeletons` to all the examples and then take the intersection of the resulting set of skeletons. Let

$$\texttt{(SkeletonIntersection P XS)} \equiv \bigcap_{X \in XS} \texttt{(Skeletons P X)}$$

`SkeletonIntersection` returns every skeletal subprocedure such that adjoining the subprocedure to the old procedure would create a procedure that is consistent with all the examples in `XS`. Section 19.1 shows that `Disjoin` cannot be defined solely as `SkeletonIntersection`. This would cause it to output skeletons that lead to star bugs. Apparently, students have some biases concerning the choice of control structures for their new subprocedures. Hypotheses are needed to capture these biases. Let `InduceSkeleton` be an undefined function to capture the control structure biases of students. In effect, it returns some subset of the skeletons returned by `SkeletonIntersection`. Given `InduceSkeleton`, the definition for `Disjoin` is

```
(Disjoin P XS) ≡
   (Adjoin P (InduceSkeleton P XS))
```

`Disjoin` outputs a set of new procedures. Each output procedure is a copy of `P` with a new skeletal subprocedure attached to it. `Adjoin` is a trivial function that attaches a set of skeletal subprocedures to `P`, producing a set of procedures Each one of these new procedures will be submitted individually to `Induce`.

Induce

Figure 16–1 shows a typical subprocedure. Rules 1 through 7 are new. They were built by `Disjoin`. Rules 8 and 9 are part of the old procedure. The new AND goal and the three trivial OR goals are new. The parent OR is old, as are the three goals labelled Kid1, Kid2 and Kid3. `Induce`'s job is to flesh out the new goals and the new rules by giving them arguments and patterns. More specifically, it has four tasks, most of which are simple bookkeeping:

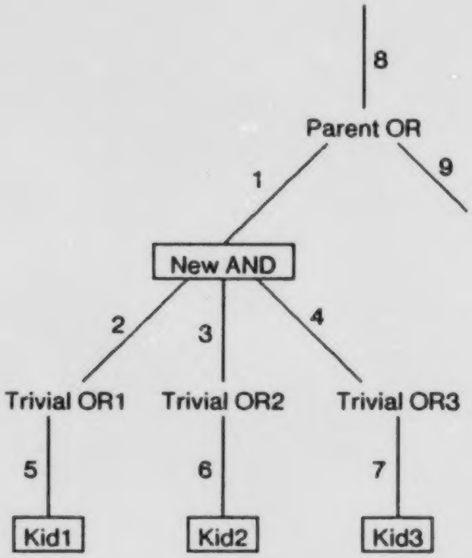


Figure 16–1
A skeletal subprocedure. Rule 1 is the adjoining rule. Goals bear generic names.

1. *Test patterns:* New OR rules need to be given test patterns. For the trivial OR rules, such as rules 5, 6 and 7 in figure 16–1, the test pattern is the null pattern, {} (recall that {} always matches, therefore it is always true). Inducing the test pattern for the adjoining rule (e.g., rule 1 in figure 16–1) is a difficult task. Let `InduceTest` be a new undefined function that calculates a test pattern for the adjoining rule. Its definition will be discussed in a moment.

2. *Fetch patterns:* New AND rules (e.g., rules 2, 3 and 4 in figure 16–1) must have their fetch patterns induced. This is another difficult induction problem. Let `InduceFetch` be a new undefined function that solves it. Its definition will be discussed in a moment.

3. *Arguments:* The new goals and the new rule's actions both need to be assigned arguments. Most of the time, this is easy. Since OR rules don't shift focus (see section 15.1), the arguments of the parent OR can be copied and used both for the adjoining rule's action arguments and for the new AND goal's arguments. Similarly, the arguments of the Kids can be copied and used both as the arguments of the trivial OR goals and as the arguments of the trivial OR rule's actions. The only arguments left are the action arguments for the new AND rules (e.g., rules 2, 3 and 4 in figure 16–1). `InduceFetch` determines these automatically: A fetch pattern represents focus shifting. The new shifted focus is bound to certain of the patterns variables. These variables are used as action arguments of AND rules. So, finding arguments for each of the goals and actions is generally just a matter of bookkeeping. There is an extra twist, however, under circumstances that are explained in the next paragraph.

4. *Functions:* If any of the Kid goals is a writing action, as is often the case, then it will require the OR rule that calls it to pass it a number or other symbol to write. Unless this symbol is a direct copy of a visible one, it must be calculated by a nest of functions. This nest is placed in the OR rule's action argument. Inducing what this function nest could be is non-trivial, so it too will be assigned an undefined place holding function, `InduceFunction`.

Out of all these bookkeeping operations, three critical tasks emerge: (1) test pattern induction, (2) fetch pattern induction, and (3) function induction. This makes some intuitive sense. When one learns a subprocedure, one first needs to discover *when* it is applicable (test pattern induction). For each action in the subprocedure, one needs to discover *where* the action should be located (fetch pattern induction) and *what* new numbers it needs, if any (function induction). These three critical tasks have been formalized as three undefined functions, `InduceTest`, `InduceFetch`, and `InduceFunction`. Defining these functions is the business of the next level, the bias level.

Practice

The function `Practice` is just like `Induce` except that it has fewer opportunities to do induction. It must use whatever the procedure has for test patterns, fetch patterns and functions in order to answer the practice problems. However, it may, in some cases, be able to narrow the space of patterns or function a little due to special characteristics of the practice problems. In general, `Practice` makes little difference in the model's predictions, so no more will be said of it.

Delete

The `Delete` function is simple to define given the AOG representation. It inputs a procedure constructed by `Disjoin`, `Induce` and `Practice`. It outputs a set of procedures. Each of the output procedures has had the rules of its new AND replaced by a proper, non-empty subset of those rules. Given a new AND with two rules, `Delete` outputs two procedures (deleting all the new AND's rules is pointless; it merely "takes back" the lesson, yielding no new predictions). Given a new AND with three rules, `Delete` outputs six procedures.

Summary

Defining the representation language allowed subprocedure induction to be almost completely defined. The above discussion sketches a formal treatment of subprocedure induction. It has been informal in places because a good deal of the subprocedure induction is tedious bookkeeping. Four new undefined functions were introduced:

```
InduceSkeleton
InduceTest
InduceFetch
InduceFunctions
```

As the names indicate, each function is an inducer. It outputs only generalizations that are consistent with the examples. However, it will soon be shown that pure induction is, in each case, too unconstrained. The functions generate generalizations that people are never observed to acquire. To get the model's predictions to match human learning, bias predicates governing each function need to be defined. That is all that is left to do. The hypotheses defining the representation are so powerful that they almost completely define the learning algorithms that the learner must use.

16.4 The solver

The previous section partially defined the learner. This does the equivalent exercise for the solver. The architectural level defined the model's overall problem solving behavior in terms of three undefined functions, whose informal definitions, repeated from chapter 8, are:

`(Interpret P S)`	represents one cycle of the normal interpretation (execution) of the procedure P. The second argument, S, is a *runtime state*: a composite of the internal (interpreter) state and the external (problem) state. **Interpret** returns the next runtime state. `Interpret` is defined by the representation language used for procedures.
`(Impasse S)`	is a predicate that is true when the runtime state S is an impasse. It is implemented by a set of *impasse conditions.* If any impasse condition is true, then **Impasse** is true. **Impasse** represents the problem detection component of local problem solving.
`(Repair S)`	represents the other half of local problem solving, problem rectification or repair. It is implemented by a set of *repairs*, such as Noop and Backup. **Repair** returns a set of states. Each state results from the action of one of the repairs on the input state S.

The aim of this section is to define these functions. It turns out that their definition will be incomplete. Two undefined functions are needed. These in turn become the target of the next level's investigation.

Runtime state

The functions use a runtime state. Now that the representation language has been defined, the runtime state can be formalized. As stated in the architecture level, the runtime state is a pair: an internal (execution) state and an external state. The external state is just a problem state. The internal state has two components, as stack and a single bit of global state, called microstate. Microstate is used to remember whether the interpreter's last manipulation of the stack was a push or a pop. Microstate is needed for formalizing the Backup repair. Backup pops the stack, which automatically sets the microstate to Pop. But Backup needs to have the interpreter *resume* the goal that Backup left on the top of the stack rather than pop it. To cause this to happen, it resets microstate to Push. This fools the interpreter, causing it to interpret the top goal as if it had just been pushed onto the stack. If microstate were not available, Backup would be less simple to formalize.

The other component of the internal state is the goal stack. The stack needs to have more than just goals in it. Each element of the stack (a stack frame) needs to be three components:

`Goal`	The name of the goal.
`Bindings`	Variable-value pairs that represent the bindings of the goal's arguments.
`ExecutedRules`	The goal's rules that have already been executed.

The need for `Bindings` follows immediately from the principle that data flow is applicative. The set of executed rules is needed so that AND goals may work properly. If the interpreter doesn't know which AND rules have already been executed, it will just execute the first AND rule over and over. For AND goals to have their intended meaning, `ExecutedRules` must be a part of the goal's state. In production systems, the same affect is achieved by the refractoriness conflict resolution principles (McDermott & Forgy, 1978).

Interpret

The `Interpret` function executes the procedure on the runtime state that it is given. It changes the state in ways directed by the procedure. It doesn't do much, only one "cycle" of interpretation. Solving a subtraction problems requires hundreds of calls to `Interpret`. Given the hypotheses on representation, `Interpret` can be almost completely defined. It can't be totally specified because pattern matching hasn't been completely defined. Two undefined functions, `Test` and `Fetch`, will be used to represent how test patterns are matched and how fetch patterns are matched:

`(Test S P)`	Given a runtime state S and a pattern P, `Test` returns true or false.
`(Fetch S P)`	Given a runtime state S and a pattern P, `Fetch` returns a set of binding sets for P's variables.

Each assignment of values to variables is a binding set. `Fetch` returns a set of binding sets because the pattern may match several ways. Or it may not match at all, in which case the set that `Fetch` returns would be empty. The local problem solver, which runs between each cycle of `Interpret`, checks for these anomalous matches and repairs them. The bias level discusses exactly what kinds of matches the local problem solver treats as anomalous. The bias level also defines exactly what the matching functions `Test` and `Fetch` actually do.

Given the two matching functions, there are many ways to define `Interpret`. One will be sketched in order to give a feel for some of the issues involved. This version of `Interpret` does either a Push or a Pop whenever it is called. That is, the cycle size is set at single pushes and pops. Finer cycle sizes are possible. For instance, each binding of a goal argument could count as a cycle of the interpreter. Although I've tried many cycle sizes for `Interpret`, none seem to have any advantages over the others.

Figure 16–2 gives the code for this version of `Interpret` and the minor functions that it employs. `Interpret` has two basic cases: (1) If microstate is Push, then the current goal has just been started up. If it is a primitive goal, then the interpreter just executes it; otherwise, a rule is chosen and executed. (2) If microstate is Pop, then the current goal has just had one of its rules executed. The choice is between resuming it, by choosing a rule and executing it, or exiting the goal by popping the stack.

The exit conventions of the interpreter are implemented by `ExitGoal?`. If the top goal is an AND, it is popped only when all its rules have been executed. If the top goal is an OR, it's done as soon as any of its rules are executed.

The conflict resolution strategies of the interpreter are implemented by `PickRule`. It makes critical use of the order of a goal's rules. AND rules are in the order that the learner saw them being executed by the worked examples. The first rule on the goal's list is the first rule executed. Because the AOG language represents all control choice as OR goals, the patterns of AND rules are not used as applicability conditions. In particular, `PickRule` just takes the next unexecuted rule on the AND's list without doing any pattern matching. For OR rules, `PickRule` tests for applicability using the undefined matching function `Test`. If more than one unexecuted OR rule is applicable, then `PickRule` returns the first one on the OR's rule list. The order of rules is used to represent the chronology of their acquisition. The most recently acquired rule is first. To summarize, the conflict resolution strategies are: (1) For AND rules, pick the first unexecuted rule. (2) For OR rules, pick the first (i.e., most recently acquired) unexecuted rule that has a true applicability condition.

A

```
If (Microstate S) = Push then
    If (Goal-TOS  S) is primitive then
        1.  (EvalGoal (Goal-TOS  S)  B  S)
        2.  (Pop S)
    else (ExecuteRule  S  (PickRule S))
  else
    If (ExitGoal? S) then (Pop S) else
        (ExecuteRule  S  (PickRule S)).
```

where

```
(ExitGoal? S) ≡
    Either
        (Goal-TOS  S) is an AND goal, and
        all of its rules are in (ExecutedRules-TOS  S),
    Or (Goal-TOS  S) is an OR goal, and
        (ExecutedRules-TOS  S) is not empty.

(PickRule S) ≡
    Return the first rule R of (Goal-TOS  S) such that
        R is not in (ExecutedRules-TOS  S) and
        either (Goal-TOS  S) is an AND goal or (Test S (Pattern R)).

(ExecuteRule S R) ≡
    1.  Add R to (ExecutedRules-TOS  S)
    2.  If (Goal-TOS  S) is an OR goal,
            then (InstantiateAction S (Action R) (Bindings-TOS  S)
            else (InstantiateAction S (Action R) (Car (Fetch S (Pattern R)))).

(InstantiateAction S A B) ≡
    1.  (Push S (ActionGoal A) {} {})
    2.  For each form F in (ActionArgs A)
            as each variable V in (GoalArgs (Goal-TOS  S))
            do   Bind V to (EvalForm F B S) and add the binding into (Bindings-TOS  S).
```

B

(Microstate S)	Returns the current setting of the microstate bit in the runtime state S.
(Goal-TOS S)	Returns the goal on the top of the stack.
(Bindings-TOS S)	Returns the bindings of the goal on the top of the stack in S.
(ExecutedRules-TOS S)	Returns the set of executed rules on the top of the stack in S.
(Pop S)	Pops the stack of runtime state S.
(Push S G RS B)	Pushes onto the stack of S a new stack frame consisting of G as the goal, RS as the set of executed rules, and B as the set of bindings.
(EvalGoal G B S)	Executes a primitive goal G in the runtime state S using bindings B. Primitive goals, e.g., Write, change the external (problem) state but do not change the internal state.
(EvalForm F B S)	Executes a form (i.e., a variable, a constant or a function) in the current state with the bindings B, and returns its value. For variables, it simply looks up the variable's binding in B. Constants are simply returned, with their QUOTE stripped off. Functions, such as the arithmetic facts functions, make no changes to S of any kind.
(Pattern R)	Returns the pattern of the rule R.
(Action R)	Returns the action of the rule R.
(ActionGoal A)	Returns the goal called by the action A.
(ActionArgs A)	Returns the list of forms that are the arguments of the action A.
(GoalArgs G)	Returns the list of variables that are the arguments of the goal G.
(Car X)	Returns the first element of a list X.

Figure 16–2

(A) Main code for Interpret. (B) Primitive and utility functions.

Focus shifting is accomplished by matching fetch patterns. Since AND rule's patterns are used as fetch patterns, the function `ExecuteRule` calls `Fetch` for AND rules but not for OR rules. Calling `Fetch` augments the current bindings with the bindings of the pattern variables. (N.B., The interpreter assumes the local problem solver will catch any mismatches, so it just uses the first element of the set of binding sets returned by `Fetch`.) When the action's goal is instantiated by `InstantiateAction`, the goal arguments are bound to the values of certain fetch pattern variables. This accomplishes focus shifting.

The point is only that the interpreter can be almost completely defined, excepting only the functions `Test` and `Fetch`, and that its definition is rather simple. The AOG language is not very complex, and neither is its interpreter.

Local Problem Solving

The local problem solver is formalized by a predicate, `Impasse`, and a function, `Repair`. Both are driven by sets. `Impasse` uses a set of impasse conditions. `Repair` uses a set of repairs. These two sets are constant parameters of the model. Although the exact membership of both sets is still open for investigation, their value, whatever it is, may not be varied across tasks or subjects. To do so would give so much tailorability to the model that the theory would be difficult to refute. Holding the sets constant represents the assertion that local problem solving is a widely known, task-independent skill. It concerns procedures *per se*, and not just procedures for solving particular kinds of tasks. Task-independence entails that impasse conditions and repairs mention only aspects of the execution state. For instance, the Noop repair simply pops the stack: it executes `(Pop S)`. The Backup repair also uses only the execution state. Its implementation is:

1. (Pop S)
2. If (Goal-TOS S) is an AND, then go to 1.
3. Set (Microstate S) to Push.

This pops the stack to the first OR goal, then resets the execution state so that it will be entered.

The impasse conditions also mention only the execution state. For instance, the following impasse condition is true if `PickRule` would fail:

if
1. (ExitGoal? S) = false, and
2. (Goal-TOS S) is an OR goal, and
3. There is no rule R in the rules of (Goal-TOS S) such that
   R∉(ExecutedRules-TOS S) and (Test S (Pattern R)) = true,

then impasse.

This condition checks for halt impasses — times when the interpreter would have to halt because no rule applies. Another impasse condition checks for mismatching fetch patterns:

if
1. (ExitGoal? S) = false, and
2. (Goal-TOS S) is a non-primitive AND goal, and
3. R is the first of its rules that is not in (ExecutedRules-TOS S), and
4. (Fetch S (Pattern R)) is not a singleton set,

then impasse.

This impasse condition checks whether the next call to `Interpret` will call `Fetch`, then it checks whether the pattern will mismatch (i.e., whether it will fail to match at all, or more commonly, whether it matches ambiguously). If so, an ambiguity impasse is signalled. The ambiguity impasse will play an important role in the Bias level.

At present, the model uses only five impasse conditions. Two have just been discussed. Another checks for infinite loops. A fourth causes an impasse when the current problem state is not syntactically well formed. The fifth impasse condition checks for precondition violations. In a sense, the precondition impasse condition is special since it must consult non-execution state information. It must look up the primitives' preconditions. Technically, this violates the principle that impasse conditions refer only to the execution state. However, preconditions are inevitable in any model that has primitives (and all computational models do). Just as the actions of a primitive operator are beneath the grain size of the model, the impasses of the primitive are also beneath the grain size. Preconditions represent internal impasses that have been lifted up to the grain size boundary. For instance, the facts function `Sub` is a primitive with a precondition. Suppose it were represented as a non-primitive procedure that, say, uses finger counting to calculate differences. It might reach an impasse calculating $5-7$ when it tries to tick off a finger and finds there are no more fingers to tick off. The precondition at the `Sub`-sized grain is a lifting of this impasse to a higher level. In short, preconditions are as inevitable as primitives. Hence, an impasse condition that refers to them is also inevitable.

The basic point is that defining the execution state allows defining the repairs and the impasse conditions. These in turn define the local problem solver.

16.5 Preview of the bias level

The hypotheses on representation have taken us a long way. They not only defined the representation language, they defined almost all of the learner, the interpreter, and the local problem solver. The only issues left to discuss concern the six undefined functions mentioned above:

```
InduceSkeleton
InduceTest
InduceFetch
InduceFunctions
Test
Fetch
```

The first four express the biases of the learner. The representational hypotheses defined all possible patterns and skeletons consistent with a lesson's examples; the bias functions filter out the choices that human learners are never observed to choose. As will be seen in the next section, these biases are *relative* rather than absolute. They compare *two* choices and say which is better. The representational hypotheses are *absolute*. In a sense, they say of a *single* choice whether or not it is good. Because the biases are relative, they cannot be built into the representation language. Representation languages can express only absolute constraints.

The other two undefined functions, `Test` and `Fetch`, concern pattern matching. There are many ways that patterns can be matched against the problem state. For instance, they can be matched to maximize the number of matched relations, or they can be matched to maximize the number of bound pattern variables. The matching issues are intimately related to the learner's bias for pattern inductions. The bias principles express how the learner views the worked problem, and the matching principles express how this viewpoint is applied to exercise problems. They are duals. They form two ends of an "informational conduit" between examples and exercises. The next level discusses both issues together, despite the fact that its name, the bias level, refers to just one.

Chapter 17
Two Patterns or One

At this point in the development of the representation, it has been shown that patterns represent the interface between the procedure and the current problem state. There are two jobs that patterns are used for: (1) shifting the focus of attention, and (2) testing applicability conditions in order to decide which rule to take. It would be parsimonious if the same patterns could be used both for testing and fetching. This is what production systems do. The pattern on the condition side of a production rule tests whether the rule is applicable, and it binds variables for use in the actions of the other side of the rule. However, it turns out that the data force the theory away from the parsimony of single-pattern rules. Two kinds are needed: *fetch* patterns and *test* patterns. This chapter discusses why both kinds are needed. The distinction between fetch and test patterns was assumed in the chapter on representational syntax; now it is time to fulfill the promise and show that the distinction is well motivated.

This chapter also introduces some important bug data that set the empirical stage for the arguments of the following two chapters. In particular, the bugs indicate some general trends for several key issues: First, the evidence indicates that induction should be biased so that fetch patterns are fairly specific. Second, it indicates that impasses occur when matching fails due to overspecific fetch patterns. Third, it indicates that induction should be biased so that test patterns are fairly general. The best way to understand these general trends is to examine the evidence itself. It concerns a group of bugs that will be called, for handy reference, the *fetch bugs.*

17.1 The fetch bugs

The fetch bugs that will be discussed here are clearly a product of incomplete learning. In particular, it seems that students were tested just after they were introduced to borrowing. Introductory borrowing is always exemplified using two-column problems, such as *a*:

$$\text{a.}\quad \begin{array}{r} 4\phantom{^{1}3} \\ 5^{1}3 \\ -\;1\;9 \\ \hline 3\;6 \end{array} \qquad\qquad \text{b.}\quad \begin{array}{r} 4\phantom{^{1}3\;7} \\ 5^{1}3\;7 \\ -\;1\;9\;2 \\ \hline 3\;6\;5 \end{array} \qquad\qquad \text{c.}\quad \begin{array}{r} 4\phantom{^{1}3} \\ 7\;5^{1}3 \\ -\;2\;1\;9 \\ \hline 5\;3\;6 \end{array}$$

There is no logical reason against using multicolumn problems, such as *b* and *c*, but in the textbooks that I've seen, they are never used in the initial borrowing lessons.

The general story for the fetch bugs goes like this: Suppose students abstract a highly specific fetch pattern to describe where borrow's decrement goes. When they are given multicolumn problems, such as *b* or *c*, their overly specific fetch patterns may not match. This triggers an impasse, leading via various repairs to each of the fetch bugs. In order to verify this story, each of the fetch bugs will be discussed in detail. The telltale Cartesian product pattern of impasses and repairs will be uncovered. However, in order to make the exposition easier to follow, it will be couched in terms of fetch patterns, impasses and repairs, just as if the point under discussion had already been decided. After the evidence is exposed, opposing hypotheses will be evaluated.

Borrow-No-Decrement-Except-Last:

$$\text{a.}\quad \begin{array}{r} 7\;5\;{}^{1}3 \\ -\,2\;1\;9 \\ \hline 5\;4\;4 \end{array}\ \times \qquad \text{b.}\quad \begin{array}{r} 4 \\ 5\;{}^{1}3\;7 \\ -\,1\;9\;2 \\ \hline 3\;4\;5 \end{array}\ \checkmark$$

When this bug's fetch pattern is induced from two-column borrow problems, the learner abstracts the fact that the column that is borrowed from is adjacent to the column that is borrowed into. The learner also abstracts that the borrow-from column is the leftmost column. This dual description is overspecific. It is true of borrow-from columns only on two-column problems. That is what ultimately leads to the bug. To make the discussion concrete, suppose that the fetch pattern contains the fragment

```
{...  (Adjacent? G BFC BIC)
      (First? G BFC)        ...}
```

where G is the variable for the problem grid, `BFC` is the column to borrow from, and `BIC` is the column to borrow into. The first relation means that the borrow-from column is adjacent to the borrow-into column. The second relation means that the borrow-from column is the leftmost column in the problem. `Adjacent?` and `First?` are always true when borrow problems are two-column problems. That is why they are present in the fetch pattern. The learner apparently chose a highly specific generalization of the two-column training examples.

On a three-column problem, such as *a*, the pattern fails to match. There is no column which is both adjacent and leftmost. This failure causes an impasse. The above bug, Borrow-No-Decrement-Except-Last, is generated by repairing the impasse with Noop. Hence, the bug just skips the decrement if the pattern doesn't match. On a three-column problem with the borrow originating in the tens column, as in *b*, the pattern matches just fine. The hundreds column is both leftmost and adjacent to the borrow-into column. The match is exact, so no impasse occurs. The decrement happens as it should. So far, the fetch bug story is born out by the bug evidence.

Always-Borrow-Left:

$$\text{a.}\quad \begin{array}{r} 6 \\ 7\;5\;{}^{1}3 \\ -\,2\;1\;9 \\ \hline 4\;4\;4 \end{array}\ \times \qquad \text{b.}\quad \begin{array}{r} 4 \\ 5\;{}^{1}3\;7 \\ -\,1\;9\;2 \\ \hline 3\;4\;5 \end{array}\ \checkmark$$

This bug is derived the same way as the one just discussed, except that the impasse is repaired differently. Instead of a Noop repair, the local problem solver uses another repair, called the Force repair (becauses it forces the interpreter to choose when there is ambiguity). The Force repair finds the closest match for the fetch pattern. When there are several closest matches, then the repair chooses one of them. In the case of problem *a*, there are two closest matches for the fetch pattern. One match binds the hundreds column to `BFC`, the column borrowed from. This binding satisfies `First?` but leaves `Adjacent?` false. The other closest match binds the tens column to `BFC`. This makes `Adjacent?` true and leaves `First?` false. When Force takes the first match, then the bug Always-Borrow-Left is generated. If it takes the second match, then the correct borrow-from placement is generated.

Two points are crucial for this bug and Borrow-No-Decrement-Except-Last: (1) The fetch pattern is too specific; it has both `Adjacent?` and `First?`. (2) Impasses sometimes occur whenever fetch patterns fail to match exactly.

Two other fetch bugs differ from the two just described only in the kind of fetch patterns they have. The new bugs' fetch patterns test the numerical relationships of the digits in the borrow-from column. In two-column borrowing problems, the tens column has the property that T>B (where T and B stand for the top and bottom digits of the column, as always). The following pattern fragment is always true for the tens column of two-column borrowing problems:

```
{... (Adjacent? G BFC BIC)
     (Part BFC T)
     (Part BFC B)
     (First? BFC T)
     (Last? BFC B)
     (LessThan? B T)
     (Not (LessThan? T B))
     (Not (Equal? T B))    ...}
```

The last three relations are the ones that matter. They specify B<T, B≤T, and B≠T. This pattern will fail to match exactly on problems that require two adjacent borrows, such as

$$\text{a.}\quad \begin{array}{r} {}^{2}\;{}^{1}4 \\ 3\;5^{1}7 \\ \underline{-\,1\;9\;8} \\ 1\;5\;9 \end{array} \qquad\qquad \text{b.}\quad \begin{array}{r} {}^{2}\;{}^{1}4 \\ 3\;5^{1}7 \\ \underline{-\,1\;5\;8} \\ 1\;9\;9 \end{array}$$

The tens column falsifies one or more of the last three relations of the fragment above. Hence, the fetch pattern will fail to match for the borrow-from for the first, units-column borrow. This failure causes an impasse. The impasse leads ultimately to the following bug:

Borrow-Don't-Decrement-
Unless-Bottom-Smaller:

$$\text{a.}\quad \begin{array}{r} 4\;{}^{1}4 \\ 5\;5^{1}7 \\ \underline{-\,1\;9\;8} \\ 3\;6\;9 \end{array}\times \qquad \text{b.}\quad \begin{array}{r} \\ 5\;5^{1}7 \\ \underline{-\,1\;5\;9} \\ 4\;0\;9 \end{array}\times \qquad \text{c.}\quad \begin{array}{r} 8\;\; \\ 5\;9^{1}7 \\ \underline{-\,1\;5\;8} \\ 4\;3\;9 \end{array}\checkmark$$

This bug results from taking the Noop repair to the impasse. It skips the decrement unless the column is T>B. That is, it impasses when the borrow-from column is not exactly like the tens column of two-column borrowing problems, with respect to the relative size of the column's digits.

Borrow-Across-
Unless-Bottom-Smaller:

$$\text{a.}\quad \begin{array}{r} 3\;\;\;\;\; \\ 4\;\;\;\;\; \\ 5^{1}5^{1}7 \\ \underline{-\,1\;9\;8} \\ 2\;6\;9 \end{array}\times \qquad \text{b.}\quad \begin{array}{r} 4\;\;\;\;\; \\ 5\;5^{1}7 \\ \underline{-\,1\;5\;9} \\ 3\;0\;9 \end{array}\times \qquad \text{c.}\quad \begin{array}{r} 8\;\; \\ 5\;9^{1}7 \\ \underline{-\,1\;5\;8} \\ 4\;3\;9 \end{array}\checkmark$$

A second bug is generated by taking the Force repair to this same impasse. The repair finds the closest matches to the fetch pattern. In problem *b*, there are just two closest matches. In problem *a*, there are three.

In problem *a*, the pattern matches the hundreds column if the adjacency relation is relaxed. This is the match which generates the bug shown above. The pattern matches the tens column if the two `LessThan?` relationships are relaxed. This match generates the correct borrow-from placement. The pattern will also match the tens column if `T` and `B` are bound, respectively, to the bottom and top digits of the tens column. That is, the match preserves the `LessThan?` relationships by "inverting" the column. This means that it will try to decrement the bottom digit, generating a very rare bug, Borrow-From-Larger (see appendix 1).

In problem *b*, where T=B in the tens, the first two matches are still available, but inverting the column is no longer a closest match. Turning the column upside down doesn't make either `LessThan?` relationship true. So, only two matches are good in problem *b*. Taking the match that relaxes adjacency gives the solution shown in problem *b*. Taking the other match generates the correct borrow-from placement. So, if the Force repair always chooses the match that relaxes adjacency, the bug shown above is generated.

Summary

The four fetch bugs described above fit into the familiar Cartesian product pattern that indicates impasses being repaired. It is summarized by the following table:

	Noop	Force
Not leftmost	B.N.D.E.L	A.B.L.
Not T>B	B.D.D.U.B.S.	B.A.U.B.S.

The rows indicate the two impasses. Row one indicates trying to borrow from a column that is not the leftmost column, and row two indicates trying to borrow from a column where $T \leq B$. The two columns indicate the two repairs, Noop and Force. The four cells of the table are abbreviations for the four bugs. This Cartesian product pattern is clear and compelling evidence for the existence of the impasses due to overspecific fetch patterns.

17.2 Test patterns ≠ fetch patterns

So far, no evidence has been presented that fetch patterns and test patterns are distinct. Although it was assumed that they were different so that the syntax of the representation could be defined in an earlier chapter, that assumption needs to be backed up. The hypotheses of the preceding chapters allow test and fetch patterns to be the same. It could be that a single pattern is used for two conceptually distinct purposes, but that the distinction is not reflected in the actual procedural representation. This single-pattern approach is the one most often used in production systems. The condition side of a production rule has patterns that serve both functions simultaneously. If the pattern matches, the rule may be run. As a by-product of the match, variables are bound for use in the action side of the rule. So production rules (and many other pattern-invocation formalisms, e.g., Microplanner, Sussman et. al, 1971) use one pattern for both testing and fetching. However, there is fairly clear evidence that two distinct patterns are needed to cover the bug data.

Single pattern and exact matching

The fetch bugs have overspecific patterns for fetching the column to borrow from. If only one kind of pattern existed, then those same overspecific patterns would be used in testing whether or not to borrow. Because the patterns are overspecific, they don't match exactly. When patterns are used for tests in production systems, truth is equated with exact matching. That is, a pattern is considered to be true only if all its relations match. Under the exact match convention, the fetch bugs' patterns are false. Since the patterns that test whether or not to borrow are false due to their overspecificity, the corresponding Borrow subprocedure won't be executed. In particular, the patterns will be false in exactly the cases where the fetch patterns were found to be causing impasses and repairs. There is a contradiction here. Because the fetch patterns are being executed, the procedure has chosen to take the Borrow subprocedure. Yet by hypothesis, the patterns governing its applicability were false. To generate the fetch bugs, either (1) the patterns are matched closely instead of the usual exact matching, or (2) test patterns are distinct from fetch patterns and furthermore, they are more general, so that they will be true (i.e., match exactly) under conditions that would cause the fetch patterns to be overspecific. The first case will be shown to be unworkable, leaving the second case as the conclusion.

Single pattern and closest match

There is a problem when closest matching is used for testing patterns. Since closest matching rarely fails, it is infeasible to use failure to represent the false value of a pattern's test. The only option is to compare the *closenesss* of matches. The rule whose pattern matches the problem state most closely is the rule to execute.

But this won't let the theory generate some of the fetch bugs. In particular, it won't generate the fetch bugs that come from the second overspecific pattern mentioned above, the one that has $T>B$, $T\geq B$ and $T\neq B$ in it. Under the single-pattern hypothesis, this pattern is used both to fetch certain locations and to test whether or not to execute the Borrow rule. Suppose a problem that is appropriate for generating the fetch bugs is presented. The pattern is matched. Because it is being used to generate the bugs, this match will not be exact. In particular, the relations $T>B$, $T\geq B$ and $T\neq B$ will fail to match. This does not make the Borrow rule inapplicable. Rather, the other rules for processing a column must have their patterns matched in order to find out which rule's pattern has the closest match. One such rule will be a rule that does ordinary, non-borrow columns. Its pattern will not match exactly. The borrow column has $T<B$ but the pattern has $T\geq B$. So the pattern relation $T\geq B$ will be false. The job of the interpreter is to decide which pattern matches more closely: the borrow pattern, which has three unmatched relations, or the non-borrow pattern, which has one unmatched relation. Note that the unmatched relation from the non-borrow pattern, $T\geq B$, is also one of the unmatched relations of the borrow pattern. Hence, any way of measuring closeness of match that is monotonic (i.e., $Q\subset P$ implies $|Q|\leq|P|$) will perfer the non-borrow pattern over the borrow pattern. Because monotonicity is widely held to be an axiom of human measures of similarity (Tversky, 1977), we can assume that the interpreter will judge that the non-borrow pattern matches more closely than the borrow pattern. Hence, the Borrow subprocedure will not be called at exactly the times when the fetch bugs show that it is being called. The theory can't generate the fetch bugs if closest matching is used to test rule applicability. The single-pattern hypothesis cannot be salvaged by postulating that closest matching be used to test for applicability. The only way to get the theory to generate the fetch bugs is to assume that Borrow's test pattern is a different pattern from its fetch pattern.

17.3 Formal hypotheses

The conclusions of this chapter are summarized in two hypotheses:

> *Two patterns*
> The representation uses different patterns for testing rule applicability and for focus shifting: Test patterns are used in choosing which OR rule to execute. Fetch patterns are used to shift the focus of attention (data flow).
>
> *Test pattern match*
> A test pattern is considered to be true if and only if it matches exactly (i.e., all its relations are true in the current problem state).

In addition to these hypotheses, the chapter yielded several general observations that will be honed in the following chapters: (1) In some cases, such as the fetch bugs, test patterns are more general than fetch patterns. (2) Fetch patterns tend to be highly specific. This can be assumed to be the result of a bias in their induction. (3) Impasses sometimes occur when fetch patterns do not match exactly due to overspecificity.

Chapter 18
Fetch Patterns

This chapter discusses fetch patterns. There are two key issues to resolve. The first concerns biasing induction: How specific are fetch patterns? At the end of pattern induction, the set of patterns that are consistent with the examples is usually very large (about 2^{100}). The patterns range from large patterns (100 relations) that are highly specific, to small patterns (one or two relations) that are highly general. The first issue of this chapter concerns which of these patterns are preferred by learners for fetch patterns. The second issue concerns a set of bugs, called the fetch bugs, which were introduced in chapter 17. They seem to be caused by fetch patterns that are overly specific. To account for the fetch bugs, the theory has to describe how their matching triggers impasses.

18.1 Version spaces

This chapter contrasts two solutions to these issues. However, before they can be stated, a new formalism must be introduced. It is a simple induction technique, dubbed *version spaces* by (Mitchell, 1982). The goal of pattern induction is to find all conjunctive patterns that are consistent with the given instances, where instances in this case are problem states. To make the discussion easier to follow, it will be assumed that the pattern being induced will be used as the test pattern for some rule. For test patterns, there are two kinds of instances, positive and negative. Positive instances are problem states that the target pattern should be true of (match exactly in). Negative instances are problem states that the target pattern should false of (should not match exactly in). To make version spaces clear, it helps to restate the pattern induction problem in logical notation.

The target pattern, call it T, should match exactly in all positive instances. Let positive instance *i* be represented by P_i, the set of all literals that are true of it. (A literal is a relation or a negated relation.) Then $P_i \rightarrow T$, where "$\rightarrow$" means logical implication. That is, T is true in problem state *i* whenever P_i implies T. Similarly, let negative instance *j* be represented by N_j, the set of all literals that are true of it. Then $N_j \rightarrow \sim T$. T is false in problem state *j* whenever N_j implies not T. So the induction problem is to find all T such that:

T is in the pattern language, and
$P_i \rightarrow T$ for all positive instances *i*, and
$N_j \rightarrow \sim T$ for all negative instances *j*.

But this is logically equivalent to finding all T such that:

T is in the pattern language and $P_i \rightarrow T \rightarrow \sim N_j$ for all *i, j*.

Implication is of course a partial order on propositions. Let's say that $X \rightarrow Y$ means that Y is *above* X in the partial order. Then P is above the P_i and below the $\sim N_j$. One may be able use an efficient representation by saving the least upper bound (LUB) of the P_i instead of the P_i, and the greatest lower bound (GLB) of the $\sim N_j$ instead of the $\sim N_j$. To make this representation equivalent to saving the states, the LUB and GLB will have to be filtered to remove elements that do not conform to the transitive implication relation just stated above. Mitchell calls this representation a *version space* (Mitchell, 1982). The filtered LUB is designated S. The filtered GLB is designated G. To put it differently:

G is the set of maximally *general* generalizations.
S is the set of maximally *specific* generalizations.

Unless the kinds of implications allowed by the pattern language are extremely limited, S and G will be enormous, making the version space representation worse than merely keeping the S_i and the $\sim S_j$. However, *in conjunctive pattern languages there is only one analytic implication:* $(X \wedge Y) \rightarrow X$. Thus, the LUB can be computed by deleting conjuncts from the P_i's descriptions (assuming that the problem state's P_i is described by the set of all relations that are true of the problem state). For instance, if P_i is $A \wedge B \wedge C$, then the only generalizations of it that are in the pattern language are:

`B∧C, A∧C, A∧B, A, B, C, true`

Because the pattern language is so strongly constrained, the logical implications are simple. Hence, the computation of the LUB is quite simple as well. The computation of G is only little more complex. One has to "filter" the GLB against S before actually computing it, so to speak. The algorithms are documented in Mitchell (1982) and Cohen and Feigenbaum (1983).

There are still the non-analytic implications to take into account, such as `(0 x) → (ID/ELT x)`. This implication states that if `x` is a zero then `x` is an identity element. It and many other non-analytic implications are implicit in the grammar. In order to dispense with computing non-analytic implications during induction, Sierra computes all the non-analytic implications on the states prior to induction. Thus, whenever `(0 G007)` occurs in a P_i or a N_j, `(ID/ELT G007)` is made to occur as well. (Actually, this falls out automatically from parsing the problem state bottom-up using the grammar.)

Constraints for the sake of efficiency

Although it is slightly off the main topic, it is worth mentioning that Sierra's implementation of version spaces makes some compromises for the sake of efficient computation. Because patterns have pattern variables, the LUB and GLB computations are NP-hard. More specifically, if patterns P and Q have n variables each, then computing their LUB is $O(n^n)$. These combinatorics reflect the usual AI matching problem: Each variable in P can be paired with any variable in Q. One way to deal with it is to use small n. Winston's blocks world inductions rarely used n larger than 5 or 6 (Winston, 1975). Using small n is impossible in Sierra's case because prediction of the data requires problem states with 10 to 30 objects, and hence the patterns have about that many variables. A second solution is to impose prior constraints on which matches will be considered. Sierra uses two constraints.

First, it is assumed that pattern variables have an implicit inequality relationship between them. That is, distinct variables must match distinct objects. Hence, when finding the LUB of two patterns, distinct variables must be mapped to distinct variables. This lowers the combinatorics to $O(n!)$. There is actually some empirical evidence for this constraint, but it won't be presented here.

Second, it is assumed that the part-whole hierarchy established by the grammar is inviolate. That is, if `x` is paired with `x'` during the LUB calculation of two pattern `P` and `P'`, and `(!Part y x)` is in pattern `P`, then `y` can only be bound to `y'` when `(!Part y' x')` is in pattern `P'`. This cuts the complexity of the matching down to $O(B!^{\log n})$, where B is the branching factor of the part-whole trees, typically about three or four. When this constraint is turned off in Sierra, an LUB computation that normally takes 30 seconds takes well over two hours. A GLB computation that normally takes a few minutes takes several days. This constraint, or something like it, is a practical necessity.

18.2 Two hypotheses

With the basics of version spaces presented, the two main issues of this chapter can be discussed: how to induce fetch patterns, and how to match them. Two hypotheses are advanced concerning the kind of inductive bias that determines fetch patterns.

1. *Topological bias:* The learner prefers *maximally specific generalizations.* That is, if <S,G> is the version space of fetch patterns, the learner chooses from the S set. The name "topological" is applied because the test for maximality is a simple topological one. (In particular, when a pattern is viewed as a labelled directed graph, one pattern generalizes another if it is a proper subgraph of the other. A maximally specific pattern is one that is not a subgraph of any larger pattern. This definition is the one used to maintain the version space.)

2. *Teleological bias:* The learner has a set of "teleological rationalizations" that are used to judge the plausibility of the various generalizations that are consistent with the examples. The bias is to choose generalizations that seem to have the most coherent rationalizations with respect to what the learner believes the purposes of the procedure are. For instance, the notion of "boundaries" might be the foundation of a teleological rationalization that requires decrements to be in the leftmost column. The rationalization goes as follows: The leftmost column is a boundary of the grid. Boundaries are often subject to special rules, in general, and especially in grid-like arrangements (e.g., checker boards, chess boards, Go boards, Asteroid fields, basketball courts, football fields, etc.). The choice of "leftmost" is rational since the tens column of a two-column problem is a boundary case. Therefore, it is especially worth noting.

The difference between the teleological and topological hypotheses is subtle. Neither hypothesis is flawless, nor do the data side convincingly with one or the other. Ultimately, the choice is made on grounds of tailorability in favor of the topological hypothesis.

The second issue discussed in this chapter concerns what kinds of fetch pattern mismatching trigger impasses. Patterns can mismatch several ways. A pattern can fail to match exactly, leaving some of its relations unmatched. A pattern can match ambiguously: There may be several ways to bind its variables to problem state objects, and each binding satisfies the whole pattern. Mismatches such as these two can be used to trigger impasses. On the other hand, the interpreter can treat them as normal events. For instance, the interpreter may not care if a few relations are unmatched if its matching convention is to take the match that maximizes the set of matched relations. This issue involves figuring out a combination of conventions for matching and triggering impasses that will maximize the coverage of the data. Two hypotheses are considered:

1. *Exact match:* The interpreter matches fetch patterns exactly. If not all of the relations of a pattern are matched, an impasse occurs.

2. *Closest match:* The interpreter matches fetch patterns closely. That is, the matcher chooses bindings for the pattern variables that maximizes the set of relations that are matched. If the fetch is ambiguous in that more than one closest match exists, then an impasse occurs.

The bias issue and the matching issue are discussed together since they turn out to be related. It will be shown that the topological bias seems the better account for the learners' bias in inducing fetch patterns. Closest matching is a better account for the use of fetch patterns during interpretation and local problem solving.

18.3 The pattern focus hypothesis

The topological hypothesis as it stands is incomplete. It needs to say something about how large an area to focus on when constructing most specific patterns. The grammar is, in principle, powerful enough to build a parse tree not only for the subtraction problem, but for the whole page containing the problems, and perhaps even larger spatial areas. Taken literally, the hypothesis that fetch patterns be as specific as possible implies that patterns mention not only the column's position in the current problem, but also the problem's position in the current page, the page's position on the desk, and so forth. In principle, fetch patterns could be infinite. Including this much context in the patterns causes the procedure to impasse whenever the problem is not set in exactly the same context as the training exercises. Clearly, this is not what people do. A constraint is needed to limit the size of the fetch patterns. The constraint will be called the *pattern focus* hypothesis.

A precise formulation of the pattern focus hypothesis can be obtained by examining the bug Always-Borrow-Left. The preceding chapter described its derivation from two-column borrowing problems. The discussion centered on the fetch pattern for borrow-from. The fetch pattern for borrow-*into* is equally interesting. The following illustrates a typical example problems' states just before each of the two actions:

```
                                        4
a.        6 3          b.             6 3
        - 1 9                       - 1 9
```

Problem state *a* is just prior to the borrow-from action. The learner induces the fetch pattern for borrow-from from states like state *a*. Problem state *b* is the sort of state used for inducing borrow-into's fetch pattern. The preceding chapter showed that borrow-from's fetch pattern is often specific enough to cause impasses on three-column problems. The essential problem was that the pattern's induction from two-column problems led to incorporating the fact that the borrow-from column is the leftmost column of the problem and it is also adjacent to the column that originated the borrow. If the induction of the fetch pattern for borrow-*into* is similarly biased, then it too should cause impasses because it expects two-column problems. But no such impasses occur. Borrow-into seems not to have the overspecific pattern that borrow-from has. This is a key piece of evidence for the pattern focus hypothesis.

Figure 18-1 shows the two fetch patterns in question prior to applying the focus pattern hypothesis. Part-whole trees are used for legibility. A part-whole tree is a way of displaying patterns that emphasizes `!Part` relationships of the pattern by drawing them as links in a tree. Pattern variables are shown as tree nodes. The node label is the variables name concatenated with the main categorical relation on that variable. For instance, `AC2` is the variable for the whole units column, which has two parts. `C2` matches the top part of the column, which contains the two digits `3` and `9`. `A2` matches a blank, the answer place for the units column. A little bit of focusing has already been applied in that the two patterns have been pruned to mention only the columns of the current problem. With no focusing at all, the patterns would be infinite. The pattern variables that are input-output variables are boxed. By "input-output variable," I mean that the variable is either

1. an *input* variable to the fetch pattern: The variable is an argument of the goal that this pattern's rule is under. Thus, this variable will be bound before pattern matching begins.
2. an *output* variable of the fetch pattern: The variable is used by the action of the rule that the fetch pattern is in. The whole point of the fetch pattern is to get its output variables bound.

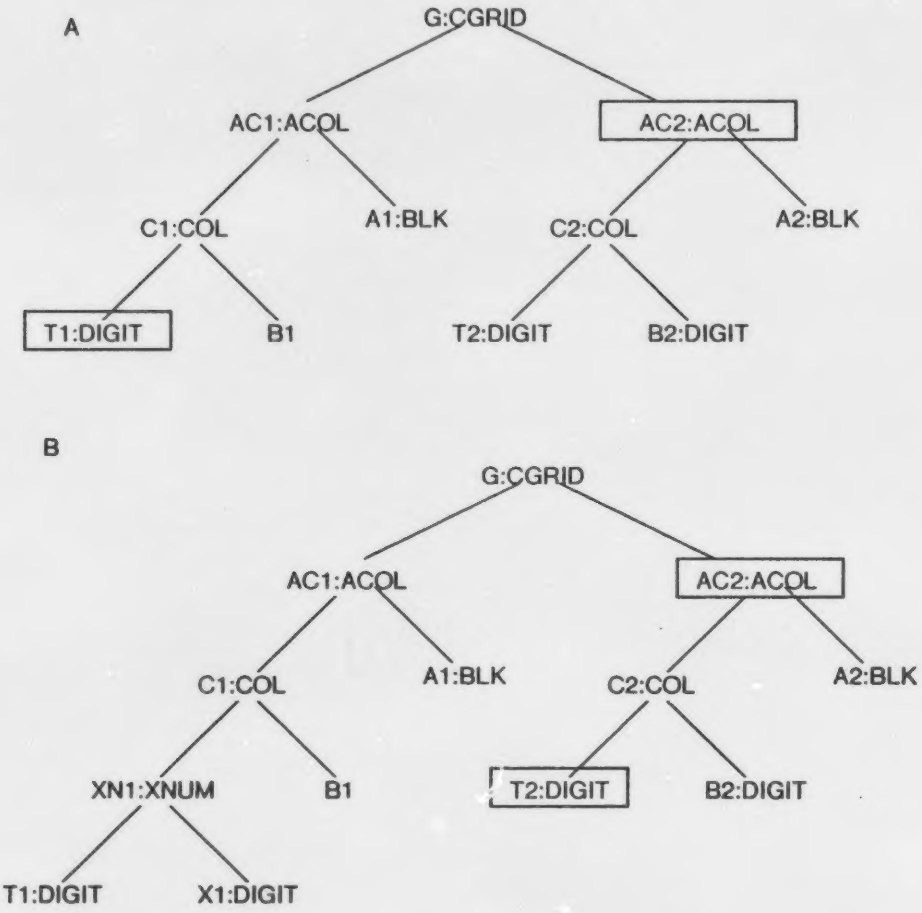

A
G:CGRID
AC1:ACOL
AC2:ACOL
C1:COL
A1:BLK
C2:COL
A2:BLK
T1:DIGIT
B1
T2:DIGIT
B2:DIGIT
B
G:CGRID
AC1:ACOL
AC2:ACOL
C1:COL
A1:BLK
C2:COL
A2:BLK
XN1:XNUM
B1
T2:DIGIT
B2:DIGIT
T1:DIGIT
X1:DIGIT

Figure 18–1
Part-whole trees displaying patterns for borrow-from (A) and borrow-into (B).

That is, if a fetch pattern is thought of as a function for retrieving objects from the problem state, then the input-output variables are the function's inputs and outputs. There are essentially just three input-output variables in the figure. `AC2` is an input variable because it is the argument of the goal Borrow in both patterns, and these patterns are on rules beneath the Borrow goal. This variable matches the whole units column. In figure 18–1a, the Borrow-from subgoal is passed the output variable `T1` as an argument. The variable matches the top digit in the tens column in problem state *a*, above. In figure 18–1b, the subgoal Borrow-into is passed the output variable `T2` as an argument. It matches the top digit of the units column in problem state *b*, above.

The only major difference between the two patterns of figure 18–1 is where the input-output variables are located. This provides the distinction that is needed to drive the pattern focus hypothesis: *A fetch pattern has the smallest part-whole tree that will span its input-output variables.* The focused fetch pattern for 18–1a is the pattern corresponding to the tree headed by `G`. The focused fetch pattern for 18–1b corresponds to the tree headed by `AC2`.

When the borrow-into pattern is focused this way, it mentions only the units column. Hence, it will not be as specific as the borrow-from pattern, which mentions both columns. In particular, the overspecificity of the borrow-from pattern is absent in the borrow-into pattern. The borrow-from pattern insists that the two columns be both adjacent and boundary columns (i.e., leftmost and rightmost, respectively). The borrow-into pattern doesn't care about adjacency since it mentions only one of the two columns. Whereas the overspecificity of the borrow-from pattern will cause an impasse, the borrow-into pattern is now general enough that it can match without causing an impasse. This explains why there are no bugs that are the equivalent of Always-Borrow-Left for borrow-into patterns. The focus pattern hypothesis has removed most of the overspecificity of the borrow-into patterns.

This formulation of the focus hypothesis makes intuitive sense. The focus of attention is no larger than the problem solver needs it to be in order to discriminate among various bindings of the output variables. If the focused pattern of figure 18–1b were larger, say including the tens column as well as the units (i.e., the focused pattern tree headed by `G`), then the tens column could potentially match several different ways without having any affect at all on the bindings of the units column section of the pattern. These matches are superfluous, given that all the solver has to do is choose the top digit of the current column, which is the units column. It is pointless to look at the tens column in order to figure out how to match the units.

18.4 Teleological rationalizations

The teleological bias is based on applying teleological rationalizations to choose among the various generalizations that induction offers. In one sense, some teleological rationalizations are already embedded in the theory. As mentioned earlier, a teleological rationalization involving boundaries (or perhaps corners) would generate the leftmost relationship that is critical to generating Always-Borrow-Left. This idea, that the ends of sequences are important, is embedded in the relationship between the grammar and the pattern relations. The spatial relation `First?` is added to patterns when the appropriate aggregate objects are sequences. However, this convention is just a choice on the theory's part. It doesn't have to be there for logical reasons. The intuitive motivation for making it a part of the grammar definition is the same as the teleological rationalization. `First?` is a defined pattern relation just because boundaries are often important and salient for sequential arrangements. There is no similar motivation for `Second?` — a relation true of the second item in a sequence — so it is not a spatial relation. The point here is that teleological rationalizations that involve spatial relationships are represented in the grammar definition.

This makes the difference between the teleological and topological hypotheses quite subtle. It essentially means that almost any spatial teleological rationalization the theorist deems necessary can be embedded in both systems. However, under the topological hypothesis, such teleological rationalizations are installed in the grammar. It can be shown that this entails that their effects will be felt more widely.

Under the teleological hypothesis, the rationalizations are in some kind of knowledge base that acts as a filter on each pattern separately. This allows the rationalizations to potentially act together, filtering individual patterns in complex ways. On the other hand, under the topological hypothesis, `First?` must be in *all* sequence-bearing patterns if it is in any. Under the teleological hypothesis, its presence can be controlled. In short, while both hypotheses allow certain teleological rationalizations to be captured, the topological hypothesis captures them in a way that introduces less tailorability into the theory.

The same difference in tailorability comes out clearly in the case of the bug Borrow-Don't-Decrement-Unless-Bottom-Smaller. The bug's derivation was discussed in chapter 17. Like Always-Borrow-Left, it has an overspecific fetch pattern. The crucial extra relationship is T>B, applied to the column to be borrowed from. This relationship is always true in two-column borrowing problems, the kind used in the initial borrowing lessons, but it is not true whenever the problem requires adjacent borrows. Hence, adjacent borrow problems cause impasses, which are what led to the inference the T>B was in the fetch pattern. One rationalization for its presence concerns the fact that the borrow-from goal has to choose between the top and bottom digits in deciding where to perform the decrement. The rationalization uses T>B as the way to distinguish the two digits on the rationalization that "the digit to make smaller is the digit that has more to begin with." Admittedly, this is not a particularly strong rationalization, but it will do to illustrate the difference in tailorability. The teleological hypothesis rationalizes T>B by using the fact that the pattern is a fetch for a *decrement*. If it was fetching for an increment, this particular rationalization would not apply. The rationalization is sensing not only the current problem state, but it is also sensing the *intended use* for the output of the fetch pattern. Clearly, this rationalization has acquired a good deal of control over whether the pattern will or will not have T>B in it. This extra power is the main difference between the teleological and topological hypotheses.

So far, I have yet to see any good use for this extra power. Any fetch that has a nice teleological rationalization is adequately derived under the topological hypothesis. Consequently, it seems wisest to base fetch pattern biasing on topological considerations. The hypothesis that captures this is:

> *Maximally specific fetch patterns*
> Given that <S, G> is the version space for a fetch pattern, `InduceFetch` chooses any pattern in S as the fetch pattern.

S stands for the set of maximally *specific* generalizations for a version space, and G stands for the set of maximally *general* generalizations. The set G is not used in the case of fetch patterns.

The pattern focus hypothesis interacts with the fetch pattern hypothesis. It limits the size of candidate fetch patterns by restricting the size of their part-whole tree.

> *Focus pattern*
> If <S, *> is the version space for a fetch pattern, and the fetch pattern is beneath goal G, and the fetch pattern will provide arguments to goal SG, a subgoal of G, then let I (for input) be the set of pattern variables corresponding to goal G's arguments and O (for output) be the set of all variables for providing arguments to SG. The part-whole tree of a pattern in S is the minimal tree necessary to span the set I∪O. Variables outside this part-whole tree are dropped from the pattern, along with all relations that mention them.

18.5 Exact matching vs. Closest matching

AI distinguishes between two kinds of conjunctive pattern matching. *Exact matching* finds bindings for the pattern variables that make *all* the pattern relations true. If there is no such binding, then matching fails. The pattern is said to be overconstrained or false. *Closest matching* finds bindings that *maximize* the subset of pattern relations that the bindings make true. Closest matching almost never fails. There is almost always some binding that makes at least one pattern element true. Closest matching actually stands for a class of matching conventions since there is some latitude in what it means for a subset of pattern relations to be *maximal.* This section discusses whether fetch patterns should be matched closely or exactly. There is a great deal of evidence that closest matching is better.

Fetch patterns need generalization long after they are acquired

When induction is biased toward specific patterns and induction is incremental, then it is inevitable that late occurring lessons will need to generalize patterns that were created earlier in the lesson sequence. The lesson that creates a pattern will do so using rather constrained, simple problems. When later lessons use more complicated problems, the simple patterns won't match exactly; they only match simple problems. To renovate the older material, the solver must either generalize the patterns so that they will now match exactly, or it must habitually use closest matching. This entailment will be clarified with an illustration.

One of the first subtraction lessons in the curriculum teaches how to do two-column problems. The students already know how to do single-column problems, such as *a* below; they are taught how to do two-column problems, such as *b*:

```
a.     6          b.     3 9
     - 2               - 1 7
       4                 2 2
```

This involves inducing several patterns. The pattern of interest is the fetch pattern that retrieves the tens column. This pattern is overspecific in the same way that Borrow-from's fetch pattern was overspecific in the generation of Always-Borrow-Left. It specifies that the retrieved column be both the leftmost column and left-adjacent column.

The evidence for closest matching appears when the students are shown their first three-column problem. Some students are able to induce that main column loop from such examples. That is, they are able to install a tail recursion into their procedure. To do so, they must install a subprocedure under one of the two currently existing column processing subprocedures. When they get done, the old column processing steps will now be called *even on three-column problems.* But on such problems, the old fetch patterns will not match exactly. The fetch patterns of the old column processing subprocedures were tuned for two-column problems. Yet now the students are processing three-column problems, presumably without local problem solving. Consequently, one must assume either that the lack of an exact match doesn't bother the interpreter (i.e., closest match is the normal interpretation of fetch patterns) or an old pattern was revised by the three-column lesson. The assimilation conjecture (section 10.1) rules out the latter position. Hence, one is left to conclude that closest matching is the usual way to interpret fetch patterns.

There are other arguments for closest matching, but they require describing the inducer in more detail than it has so far been described. Suffice it to say that closest matching is required when the old procedure is used to "parse" the problem state sequence of the worked example exercise (see section 19.1 on skeletons and finding them).

Closest matching is what Sierra uses. I have found no empirical problems with it. The only drawback is the inconvenience caused by the fact that it is roughly an order of magnitude slower than exact matching (closest matching is an NP-hard problem; in fact, it can use the same algorithm as the calculation that finds the LUB of two patterns). These considerations motivate the following hypothesis:

> *Fetch pattern matching*
> Let b be a binding of the fetch pattern's variables to objects of the current problem state such that all input variables have their correct bindings. Let P_b be the subset of relations in fetch pattern P that are true (satisfied) by b. Then b is a valid match for the fetch pattern only if (1) there is no b' such that $P_b \subset P_{b'}$ and $P_b \neq P_{b'}$, and (2) all the part-whole relations of P are in P_b.

The last clause of the hypothesis says that part-whole relations can't be relaxed by closest matching. This stipulation tames much of the combinatorial complexity of the match. It reduces complexity from O(v!) where v is the number of pattern variables in P, to $O(B!^n)$, where B is the branching factor of the part-whole tree and n is the number of non-leaf nodes in the part-whole tree.

18.6 When does mismatching trigger repair?

Having determined that closest matching is the normal way to match fetch patterns, the question of triggering impasses can be quickly dispatched. In chapter 17's discussion of the fetch bugs, it was shown that some kind of matching failure is triggering local problem solving. This was shown by the existence of Noop repairs (i.e., the bugs Borrow-Don't-Decrement-Unless-Bottom-Smaller and Borrow-No-Decrement-Except-Last) alternating with Force repairs (i.e., the bugs Always-Borrow-Left and the compound bug Borrow-Skip-Equal & Borrow-Skip-Top-Smaller).

One potential candidate for triggering is that the fetch patterns fail to match exactly. But clearly, if closest matching is the norm, this cannot be the impasse condition.

When the fetch patterns of the fetch bugs are matched closely, it turns out that they each match ambiguously. That is, there is more than one way to bind pattern variables to objects in the current problem state. Moreover, in each case, the bindings of the *output* variables are ambiguous. For instance, the pattern matcher might report back that either the tens or the hundreds column would be okay for borrowing from. This ambiguity seems to be the reason for calling in the local problem solver. To put it intuitively, if the interpreter can't decide between several ways to bind a subgoal's arguments because the fetch was ambiguous, then an impasse occurs. This discussion motivates the following hypothesis:

> *Ambiguity impasse*
> If a fetch pattern matches ambiguously so that the output variables have two or more distinct bindings, then an ambiguity impasse is triggered.

18.7 Summary

There is a theme which unifies these disparate results concerning fetch patterns. The basic problem that fetch patterns solve is disambiguating which of the many visible objects an action should use. Given this charter, it makes sense that it is only when the fetch pattern fails to disambiguate an output variable that the solver decides that something is wrong and therefore some local problem solving is called for. This story provides intuitive motivation for the ambiguity impasse hypothesis. The theme of disambiguation offers an explanation for the other hypotheses as well.

In order to maximize the fetch pattern's power to discriminate, the learner remembers everything about the lesson's examples that might prove useful in fetching. It does so in order that problem solving can approximate the lesson situation as closely as possible when it chooses bindings for the fetch pattern. That is, the learner remembers the most specific fetch patterns possible. This motivates the maximally specific fetch patterns hypothesis and the fetch pattern matching hypothesis.

However, the learner leaves behind detail if it can be assured that the omitted information won't help the solver do disambiguation of the output variables. Hence, it leaves behind information from "distant" parts of the pattern since such information is only weakly related to disambiguating the output variables. This motivates the focus pattern hypothesis.

Chapter 19
Test Patterns and Skeletons

Several related issues are discussed in this chapter. The first concerns test patterns. As with fetch patterns, inductive learning is usually not sufficient to uniquely specify a test pattern because the examples used in lessons are not variegated enough. A typical lesson would leave an unbiased inducer to decide among several million test patterns. This would generate a much wider variety of bugs than are observed. Apparently, human learners have some bias in their choice of test patterns. One of this chapter's topics is discovering that bias and making it precise. Two hypotheses are proposed and contrasted.

1. The *topological* hypothesis is that learners choose test patterns that are the maximally general generalizations of the possible test patterns. That is, if <S,G> is the version space of test patterns, the learner chooses from the G set. The name "topological" is applied because the test for maximality is a simple topological one. (In particular, one pattern generalizes another if it is a proper subgraph of the other (see section 18.1). A pattern is a maximally general pattern if there is no smaller pattern that is a subgraph of it. This definition is the one used to maintain the version space.)

2. The *teleological* hypothesis is that the learner has a set of teleological rationalizations that sanction only test patterns that fit into common sense, general notions of what the purposes of steps typically are. For the sake of discussion, teleological rationalizations are formalized as step schema (Goldstein, 1974), which expresses the general form and purpose of archetypical steps. For instance, one particularly important step schema is the *preparation step schema*: it rationalizes a step as having the purpose of preparing for some existing "main" step. The preparation step schema constrains the choice of test pattern. It might force the test pattern to incorporate a precondition for some action in the main step since avoiding a precondition violation is one purpose for a preparation step. The learner's teleological rationalizations about test patterns are represented by a set of step schemata.

Another issue discussed in this chapter concerns inducing the skeletons of new subprocedures. The skeleton of a new subprocedure is its goals and rules, stripped of the goal arguments, patterns and action arguments. Choosing a skeleton for the new subprocedure is `InduceSkeleton`'s job (see section 16.3). The choice of skeleton is totally determined by choosing the parent OR goal for the subprocedure and the set of subgoals that the subprocedure will call (called the subprocedure's kids). Thus, the issue to be discussed is twofold: (1) under which goal in the existing procedure should `InduceSkeleton` attach the new subprocedure, and (2) which goals should `InduceSkeleton` have the new subprocedure call? Induction often leaves several choices open.

Human learners exhibit distinct biases in their choice of subprocedure skeletons. The two hypotheses concerning test pattern bias are extended to cover skeleton bias. The topological bias, which has the learner choose maximally general test patterns, is extended to bias the learner to choose a place for the subprocedure that will make the new subprocedure be as small as possible. Later it will be shown how minimizing the size of a new subprocedure increases its generality. The teleological hypothesis is that step schemata control the placement of steps.

19.1 The topological bias hypothesis

To see what skeleton induction involves, a computer science fixture is needed: the trace of a procedure's execution. A trace is a tree erected over a problem state sequence that shows the program's subroutine calling history. The usual way to generate a trace is by placing "print" statements just before and just after subroutine calls. This generates a long, printed listing. A trace can also be presented as a tree. Figure 19–1 shows the trace tree for a correct subtraction procedure (in fact, the one of figure 2–6) solving a BFZ (i.e., borrow from zero) problem. Each call is shown as a tree node, with its arguments abbreviated. A trace tree is just a parse tree for the problem state sequence, using the procedure as the grammar. This is a fundamental concept in the following discussion. Parsing problem state sequences defines which skeletons exist. Seeing what a skeleton is becomes simple now that trace trees have been introduced. If the procedure is missing the BFZ goal, then the trace tree would have a hole in the middle of it, as in figure 19–2. The gap is right where the BFZ node would be. From the figure, one can see that a skeleton can be characterized by the link coming into it from above, and the links leaving it from below. Since a skeleton represents where in the parse tree a new subprocedure should go, it's no surprise that its defining aspects are topological. More formally, a skeleton is uniquely specified by a pair:

`Parent`: The name of the OR goal that the new subprocedure will be attached to.

`Kids`: An ordered list of goal names. These will become the actions on the AND goal rules of the new subprocedure. Note that only an action's name and not its arguments appear here.

Almost all problem state sequences, including the example of figure 19–2, admit more than one parent-kids pair. Most of the ambiguity is due to the fact that one can almost always make a skeleton bigger. The kids can be lower in the tree; the parent can be higher. Figure 19–3 and 19–4 show some skeletons for the BFZ problem state sequence. Figure 19–3 has lower kids. Figure 19–4 has a higher parent. Any node that would complete an otherwise incomplete trace tree is a legitimate skeleton.

When a skeleton is expressed by <parent, kids>, then the hypotheses of the theory entail that subprocedure placement can be partially solved simply by intersection of these pairs. That is, for each example in a lesson, there is a set of possible skeletons; the learner takes the intersection over these sets. Because a lesson may introduce just one subprocedure, all the skeletons' parents must be the same. Because the new subprocedure is disjunction-free, each skeleton's list of kids must be equal to each other skeleton's list of kids. In particular, two kid lists (A B C) and (A D C) cannot be merged by using disjunction on the middle kid to form something like (A (OR B D) C).

Skeleton intersection is powerful enough that it is sometimes possible to devise a lesson that yields a unique skeleton when its example's skeletons are intersected. However, for some lessons, this is not possible. In fact, the BFZ skeleton that is our running illustration cannot be uniquely specified by examples. The proof relies on the distribution of branches in the AOG that is input to the learner. The AOG is shown in figure 19–5. If the examples are designed to exemplify BFZ, then they will always allow the skeleton to have `BORROW/FROM` as parent. However, because `BORROW/FROM` is the first subgoal of `REGROUP`, a second skeleton is always possible. It is a skeleton whose parent is `1/BORROW`. This skeleton is shown in figure 19–4. It will always be legal no matter what the example. So, the intersection of skeletons over all examples will always have both skeletons.

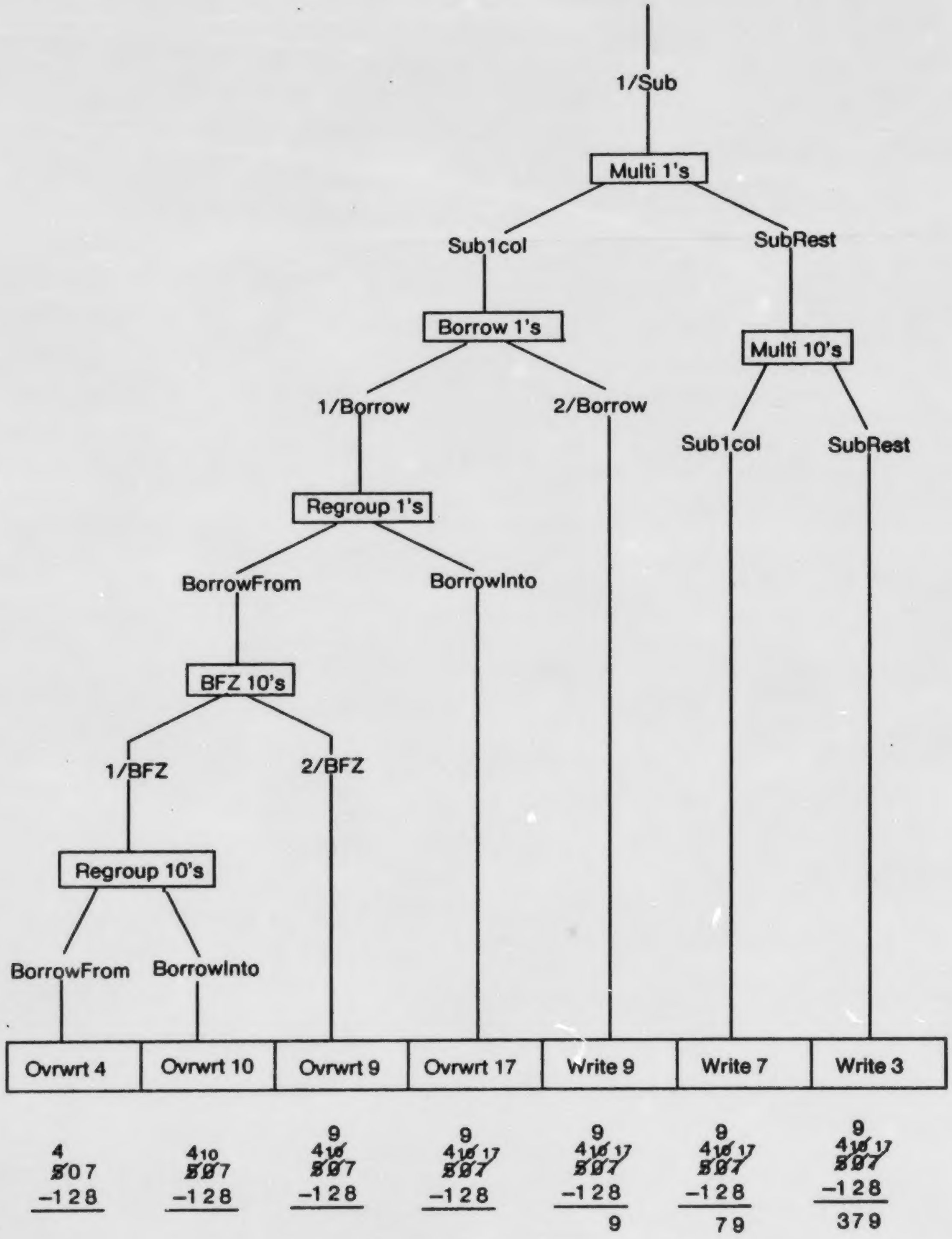

1/Sub
Multi 1's
Sub1col
SubRest
Borrow 1's
Multi 10's
1/Borrow
2/Borrow
Sub1col
SubRest
Regroup 1's
BorrowFrom
BorrowInto
BFZ 10's
1/BFZ
2/BFZ
Regroup 10's
BorrowFrom
BorrowInto
Ovrwrt 4
Ovrwrt 10
Ovrwrt 9
Ovrwrt 17
Write 9
Write 7
Write 3
507
−128
9
79
379

Figure 19–1
Trace tree for solution of a BFZ problem.

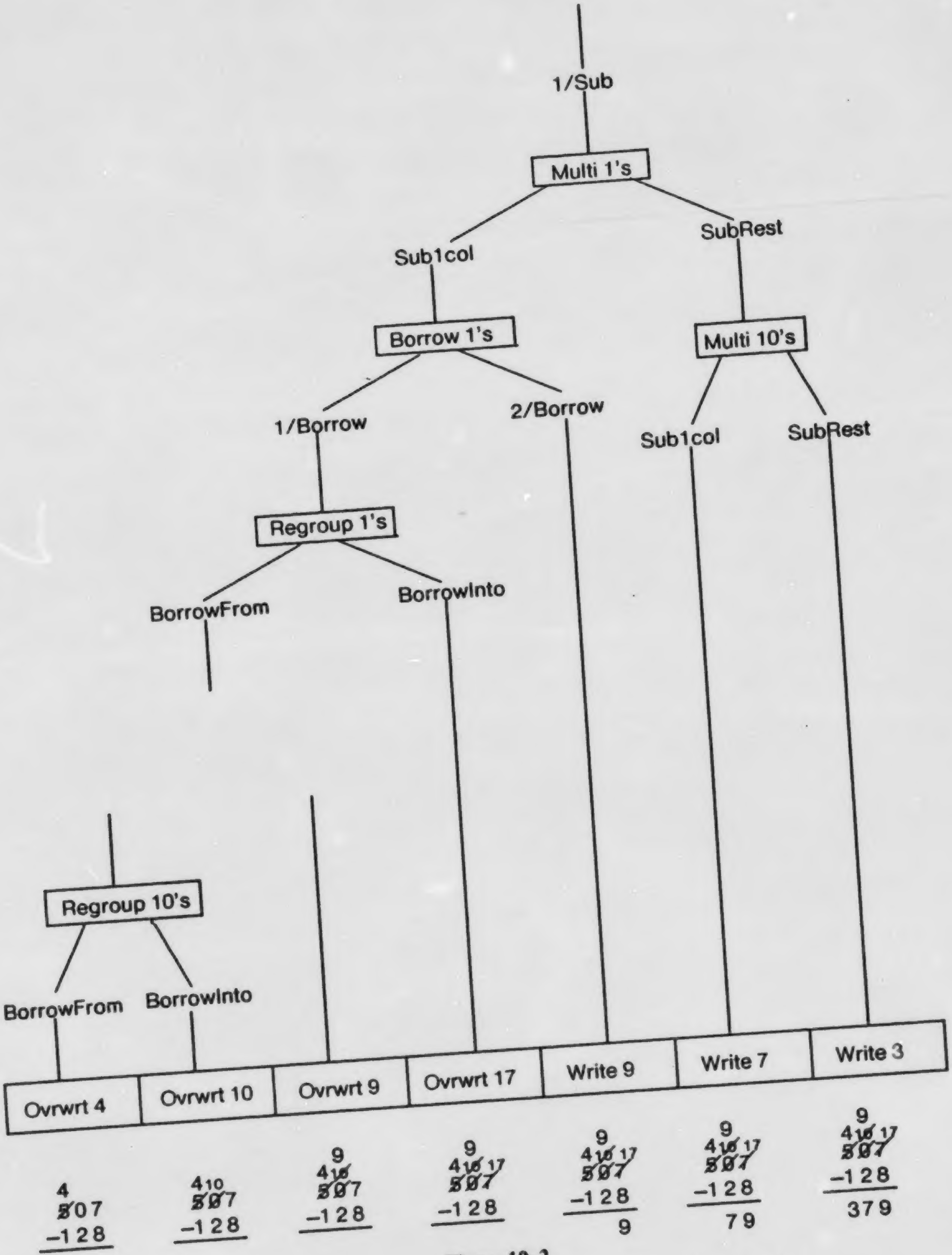

1/Sub
Multi 1's
Sub1col
SubRest
Borrow 1's
Multi 10's
1/Borrow
2/Borrow
Sub1col
SubRest
Regroup 1's
BorrowFrom
BorrowInto
Regroup 10's
BorrowFrom
BorrowInto
Ovrwrt 4
Ovrwrt 10
Ovrwrt 9
Ovrwrt 17
Write 9
Write 7
Write 3
-128
9
79
379

Figure 19–2

The skeleton is right where the BFZ node and its daughters were.

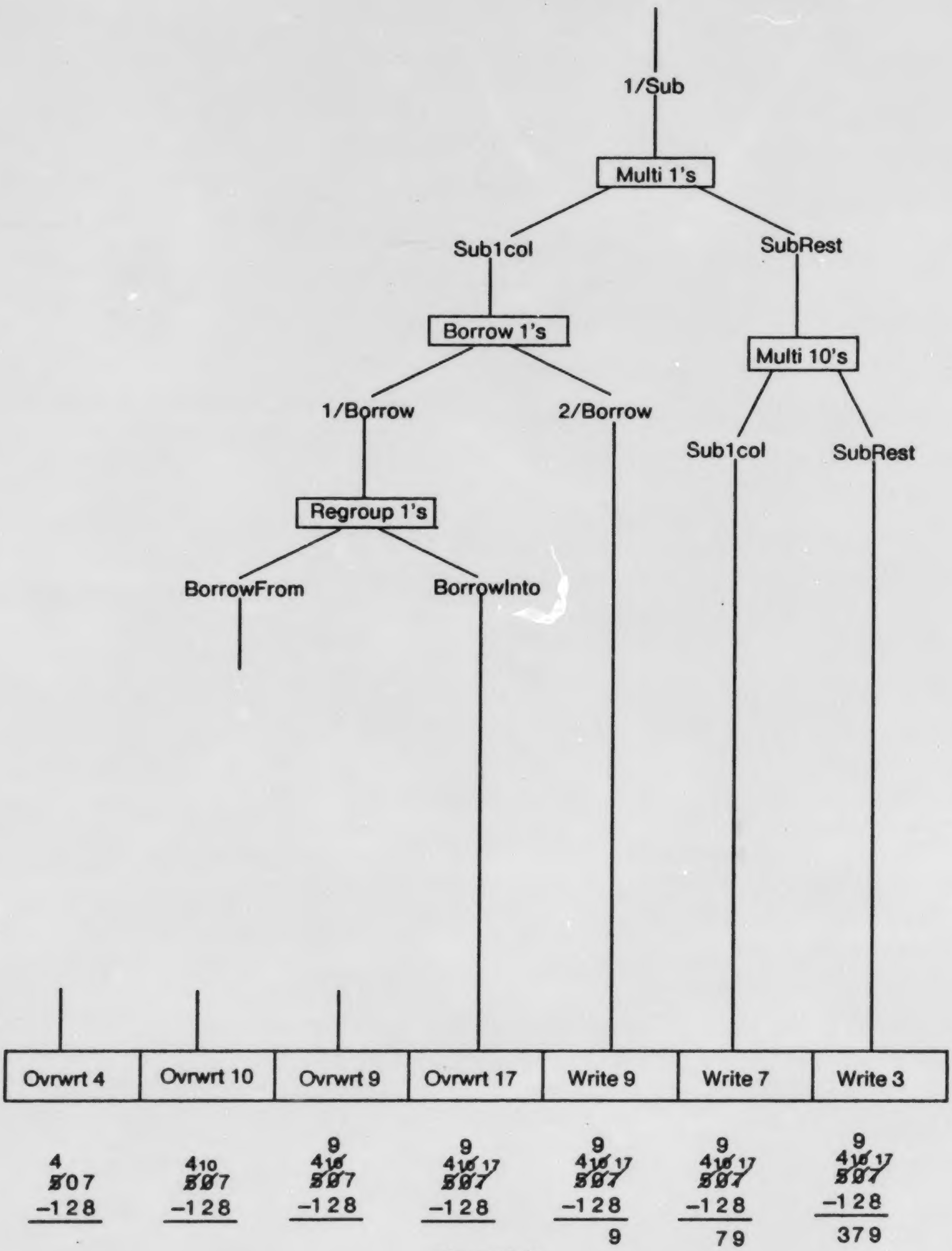

1/Sub
Multi 1's
Sub1col
SubRest
Borrow 1's
Multi 10's
1/Borrow
2/Borrow
Sub1col
SubRest
Regroup 1's
BorrowFrom
BorrowInto
Ovrwrt 4
Ovrwrt 10
Ovrwrt 9
Ovrwrt 17
Write 9
Write 7
Write 3
507
−128
9
79
379

Figure 19–3
The kids of the skeleton can be lower.

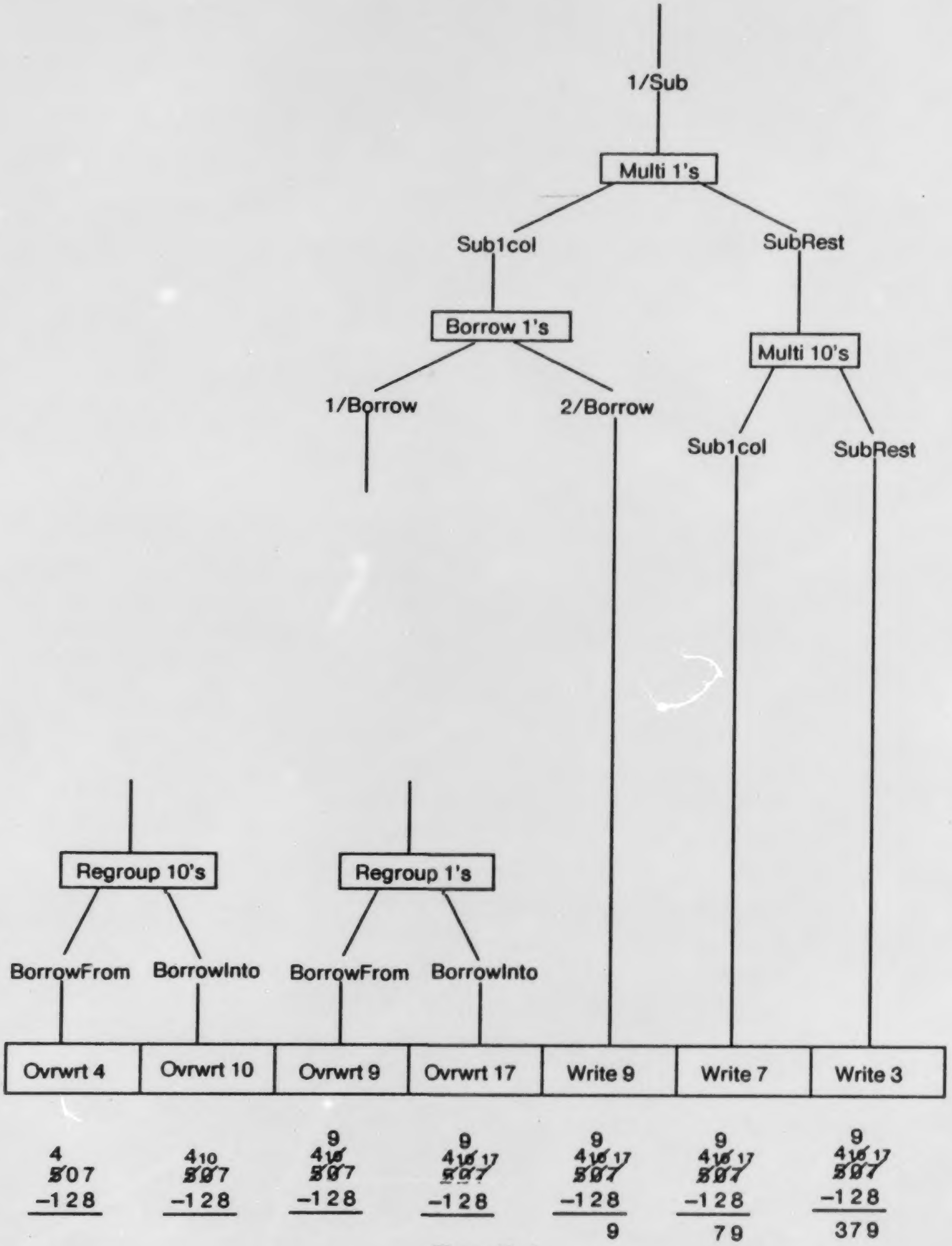

1/Sub
Multi 1's
Sub1col
SubRest
Borrow 1's
Multi 10's
1/Borrow
2/Borrow
Sub1col
SubRest
Regroup 10's
Regroup 1's
BorrowFrom
BorrowInto
BorrowFrom
BorrowInto
Ovrwrt 4
Ovrwrt 10
Ovrwrt 9
Ovrwrt 17
Write 9
Write 7
Write 3
507
−128
9
79
379

Figure 19–4
The parent of the skeleton can be higher.

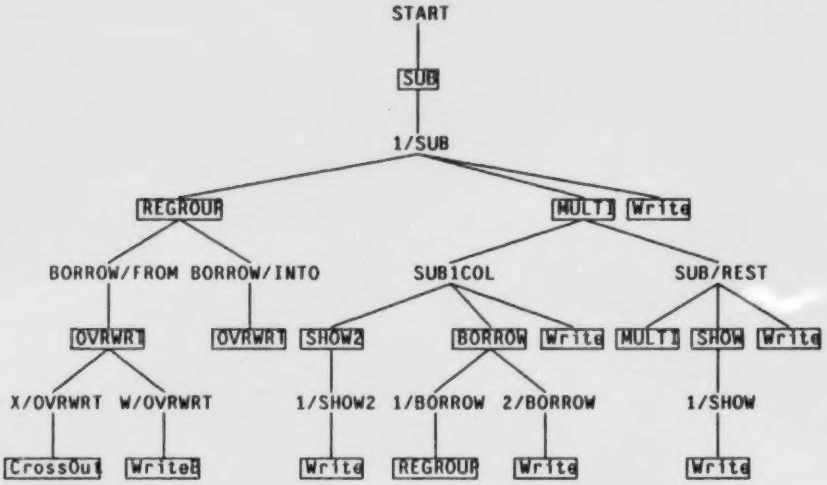

START
SUB
1/SUB
REGROUP
MULTI
Write
BORROW/FROM
BORROW/INTO
SUB1COL
SUB/REST
OVRWRT
OVRWRT
SHOW2
BORROW
Write
MULTI
SHOW
Write
X/OVRWRT
W/OVRWRT
1/SHOW2
1/BORROW
2/BORROW
1/SHOW
CrossOut
WriteB
Write
REGROUP
Write
Write

Figure 19–5
AOG for a subtraction procedure that doesn't know how to BFZ.

Perhaps even more telling than this result is the fact that textbooks often do not vary the examples enough that skeleton intersection would converge even when the AOG topology would permit such convergence. For instance, many textbooks introduce BFZ using only three-column problems despite the fact that students often know how to handle four-column subtraction already. When only three-column problems are used, a skeleton whose parent is `1/SUB` (see figure 19–5) also survives skeleton intersection. That is, the learner can't tell whether BFZ is a prefix to the whole of the subtraction procedure, or only a prefix to one column's processing. If this skeleton at the root level is not somehow filtered out by the theory, then it will survive to predict a star bug (the star bug can borrow-from-zero only when the BFZ originates in the units column). In short, something other than skeleton intersection, i.e., the learner's skeleton bias, is shouldering quite a bit of the learning load.

Potential generality

The problem of skeleton induction is a general problem for learners of structural concepts. Iba's arch learner is a good illustration of the problem in a familiar domain (Iba, 1979). Iba's arch learner doesn't know about `PRISM`, the disjunction of `BRICK`, `WEDGE`, and a few other block types. Lacking this prefabricated disjunction, when the learner sees the appropriate examples for inducing that the lintel of an arch is a `PRISM`, it must create a disjunction. The disjunction that Iba chooses is:

```
a.   (OR (AND   (ISA LINTEL 'BRICK)
                (ISA LEG2 'BRICK)
                (ISA LEG1 'BRICK)
                (SUPPORTS LEG1 LINTEL)
                (SUPPORTS LEG2 LINTEL))
         (AND   (ISA LINTEL 'WEDGE)
                (ISA LEG2 'BRICK)
                (ISA LEG1 'BRICK)
                (SUPPORTS LEG1 LINTEL)
                (SUPPORTS LEG2 LINTEL)))
```

This concept is just the disjunction of the brick-lintel arch's description and the wedge-lintel arch's description. However, Iba could have implemented the learner to construct a different disjunction:

```
b.   (AND (OR   (ISA LINTEL 'BRICK)
                (ISA LINTEL 'WEDGE))
          (ISA LEG2 'BRICK)
          (ISA LEG1 'BRICK)
          (SUPPORTS LEG1 LINTEL)
          (SUPPORTS LEG2 LINTEL)))
```

This concept disjoins only the type of the lintel. It is logically equivalent to the other concept. Neither is more general than the other. They have exactly the same extensions. However, the smaller disjunction, *b*, is more easily generalized in the future. Suppose Iba wanted to generalize the types of the legs from `BRICK` to `PARALLELEPIPED` (a solid with two faces that are parallel and the same shape, e.g., cylinders and prisms). If Iba's examples used cylindrical legs and brick lintels, then *a* and *b* would become, respectively, concepts *c* and *d*:

```
c.   (OR (AND   (ISA LINTEL 'BRICK)
                (ISA LEG2 'PARALLELEPIPED)
                (ISA LEG1 'PARALLELEPIPED)
                (SUPPORTS LEG1 LINTEL)
                (SUPPORTS LEG2 LINTEL))
         (AND   (ISA LINTEL 'WEDGE)
                (ISA LEG2 'BRICK)
                (ISA LEG1 'BRICK)
                (SUPPORTS LEG1 LINTEL)
                (SUPPORTS LEG2 LINTEL)))

d.   (AND (OR   (ISA LINTEL 'BRICK)
                (ISA LINTEL 'WEDGE))
          (ISA LEG2 'PARALLELEPIPED)
          (ISA LEG1 'PARALLELEPIPED)
          (SUPPORTS LEG1 LINTEL)
          (SUPPORTS LEG2 LINTEL)))
```

In concept *c*, only the first disjunct has been generalized. The examples used only brick lintels, so there is no evidence that the wedge-lintel disjunct should be generalized. Consequently, concept *c* will not match an arch with cylindrical legs and a wedge lintel, but concept *d* will. The *d* conception of the arch is more general. Moreover, its generality is due *only to the placement of the earlier disjunction*, because the same inductive biases and the same examples were used to generate *c* and *d* from their predecessors. To coin a phrase, concept *b* has more *potential generality* than concept *a*. Given two concepts that are equivalent in generality, one is a potential generalization of the other if there exists an inductive algorithm and a set of training examples that makes the first concept more general than the second concept.

Biases for subprocedure placement

It is not hard to see that in an And-Or language, such as Iba's representation of arches or Sierra's AOGs, choosing a smaller disjunction gives the resulting concept greater potential generality. The precise definition of "smaller" depends on the representation language. The skeleton of figure 19–4 yields a procedure with greater potential generality than the skeleton of figure 19–4. These considerations of potential generality motivate the hypotheses which define the maximal generality bias for subprocedure placement:

> *Lowest parent*
> Given two subprocedures, A and B, for possible addition to a procedure P, if A is *lower* than B in that there is a path from P's root to A that passes through B, then `InduceSkeleton` chooses A.
>
> *Fewest kids*
> If two subprocedures, A and B, have the same parent OR, and A has fewer kids than B, then `InduceSkeleton` chooses A.

As the name "lowest parent" indicates, the first hypothesis biases the learner to choose skeletons whose parents are low in the parse tree. This hypothesis chooses the skeleton of figure 19–2 over the skeleton of figure 19–4 because `BORROW/FROM` is lower than `1/BORROW`.

The fewest kids hypothesis biases the learner to choose actions for the new subprocedure that make maximal use of the old procedure. For illustration, compare the kids of the skeleton of figure 19–2 with the kids of the skeleton in figure 19–3. The latter skeleton is also a possible parse of the BFZ example's problem state sequence. It has three kids. Note that it does not have `REGROUP` as a kid. This means that the new subprocedure constructed from this subprocedure will not be recursive. It won't be able to borrow across multiple zeros. The skeleton of 19–2 can. Hence, the skeleton of 19–2 is already more general than the skeleton of 19–4. This illustrates how the fewest kids hypothesis implements a bias towards maximal generality.

The fewest kids hypothesis often leaves an important choice unmade. In an AOG, every AND has an OR just above it. Hence, whenever an AND occurs as a kid in a skeleton, it will always be possible to use the OR instead. For instance, the skeleton of figure 19–2 has `REGROUP` and `OVRWRT` as kids. It could equally well have the OR nodes just above them as kids. In fact, there are four possible kids:

1. `1/BORROW` `BORROW/FROM`
2. `REGROUP` `BORROW/FROM`
3. `1/BORROW` `OVRWRT`
4. `REGROUP` `OVRWRT`

All these choices are legal with respect to the lowest parent hypothesis and the fewest kids hypothesis.

The maximal generality bias would advise taking the highest nodes, namely choice 1. Surprisingly, this is not what students do. This particular procedure does not offer a good example, but the character of the evidence can be indicated Suppose `BORROW/FROM` is chosen as the second kid because it is higher than `OVRWRT`. This choice makes BFZ recursive, because the skeleton parent is also `BORROW/FROM`. (Actually, because this particular procedure has `REGROUP`, BFZ would be recursive anyway. Some core procedures lack `REGROUP`, because they are taught with lesson sequences that do not have special regrouping lessons, such as lesson sequence HB, which was discussed in chapter 2. For such core procedures, BFZ will only be recursive after this single-zero BFZ lesson if `BORROW/FROM` is chosen as the second kid.) Thus, after a lesson on BFZ that uses only single zero problems, such as *a*,

$$\text{a.}\ \begin{array}{cccc} & 2 & 9 & \\ & 3^1 & 0^1 & 5 \\ - & 1 & 2 & 8 \\ \hline & 1 & 7 & 7 \end{array} \qquad \text{b.}\ \begin{array}{ccccc} & 2 & 9 & 9 & \\ & 3^1 & 0^1 & 0^1 & 2 \\ - & 1 & 2 & 3 & 8 \\ \hline & 1 & 7 & 6 & 7 \end{array} \qquad \text{c.}\ \begin{array}{ccccc} & 2 & & 9 & \\ & 3^1 & 0^1 & 0^1 & 2 \\ - & 1 & 2 & 3 & 8 \\ \hline & 1 & 8 & 6 & 7 \end{array} \qquad \text{d.}\ \begin{array}{ccccc} & & & 9 & \\ & 3 & 0^1 & 0^1 & 2 \\ - & 1 & 2 & 3 & 8 \\ \hline & & & & 7 \end{array}$$

the student can do multi-zero BFZ, as shown in *b*. However, there are several bugs that indicate students do not always achieve such competence from the single-zero lesson. One bug's work is shown in *c*. It does not know how to borrow from multiple zeros. Hence, when it does the problem, it winds up trying to decrement the zero in the hundreds column. This violates a precondition: zero cannot be decremented. The solver impasses and repairs with Noop. At the end of the units column, the problem looks like *d*. This bug is called Stops-Borrow-At-Multiple-Zero. There are a few other bugs like it resulting from different repairs to the same impasse. The existence of these bugs shows that not all students acquire a recursive BFZ from single-zero BFZ lessons. In order to generate these bugs, the maximal generality bias cannot be used to resolve the four-way kids choice mentioned above. There are similar illustrations involving the main column traversal loop (i.e., if a maximally general kid is chosen, then the learner acquires the loop too soon). These facts motivate the following hypothesis:

> *Lowest kids*
> If two subprocedures, A and B, have the same parent OR and the same number of kids, and each of A's kids is lower than or equal to the corresponding kid in B, then `InduceSkeleton` chooses A, where "lower" is defined as in the lowest parent hypothesis.

Given the four-way choice mentioned above, this hypothesis has the learner pick choice 4 as the skeleton's kids.

It is not entirely clear why this hypothesis exists, given the learner's general bias toward maximal generality. It seems that the maximal generality bias applies only to the choice of *when* new subprocedures are executed and not to what the new subprocedure's subgoals will be. The choice of subgoals (= kids) is perhaps governed by the same general bias as the choice of fetch pattern. Fetch patterns are chosen to maximize their specificity. The lowest kid hypothesis also maximizes specificity.

Test pattern bias

Having discussed the biases relevant to skeletons, it is time to turn to the other half of inducing the "when" part of new subprocedures: test pattern induction. The maximal generality bias is easily formalized to apply to test patterns:

> *Maximally general test patterns*
> Given that ⟨S,G⟩ is the version space for a test pattern, `InduceTest` chooses a g∈G as the test pattern.

19.2 Step Schemata

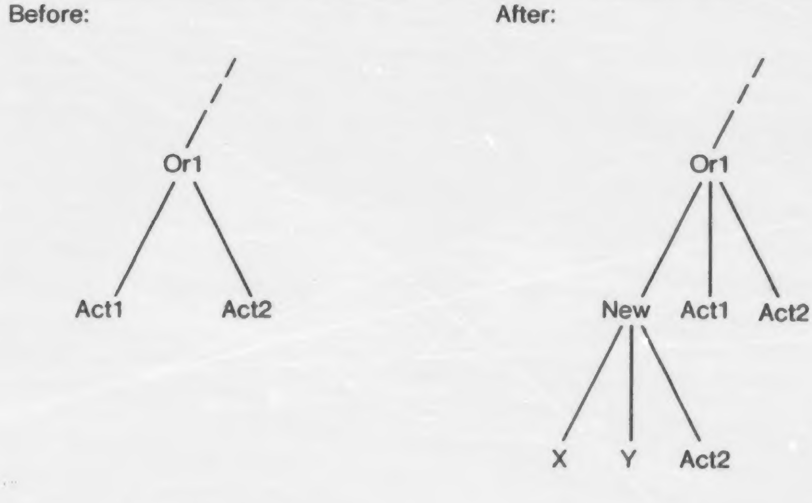
There is a completely different approach to choosing skeletons and test patterns. It postulates teleological knowledge of the student. Instead of using generality as the criterion, the learner selects only those subprocedures that can be recognized as instances of general teleological rationalizations. For the sake of discussion, these rationalizations will be taken to be step schemata (Goldstein, 1974).

Several step schemata will be presented before discussing the overall quality of the approach.

Three step schemata

Perhaps the most important step schema is the *preparatory step schema.* It matches skeletons that insert some material just before an existing action, presumably to insure that the action's preconditions are met. Technically, the recognition pattern for the preparatory step schemata merely checks that the last kid in the skeleton's kid list is a goal that will wind up being a sister to the goal of the new subprocedure. This is easier to understand with the aid of figure 19–6. The preparatory step schema applies only if the new subprocedure prepares for an existing action. Thus, if 19–6a is a fragment of the learner's input AOG, then the new subprocedure will have to have either `Act1` or `Act2` as its last kid. In the resulting procedure (19–6b), the new AND goal (**`New`**) has a few preparatory actions (`X` and `Y`), then the action being prepared for (`Act2`). Borrowing could be acquired as a preparatory subprocedure for the main column processing operation. Let Diff abbreviate that operation, which is `(Write A (Sub (Read T) (Read B)))`. So Borrow is a preparatory subprocedure for Diff. Similarly, BFZ could be acquired with a preparatory step schema. In both cases, the teleology of preparation is semantically correct. This is not in general true of the application of step schemata. Sometimes the purpose attributed to the acquired subprocedure only seems correct to the student, when in fact the correct purpose is completely different.

The *cleanup step* schema is the dual of the preparatory step schema. It matches skeletons that insert material just *after* an existing action. Goldstein (1974) calls the preparatory step and the cleanup step schemata *interface steps* since they take care of the details of meshing a main step into a sequence of other steps.



(a) (b)

Figure 19–6
Fragments of an AOG before and after preparatory step schema has applied.

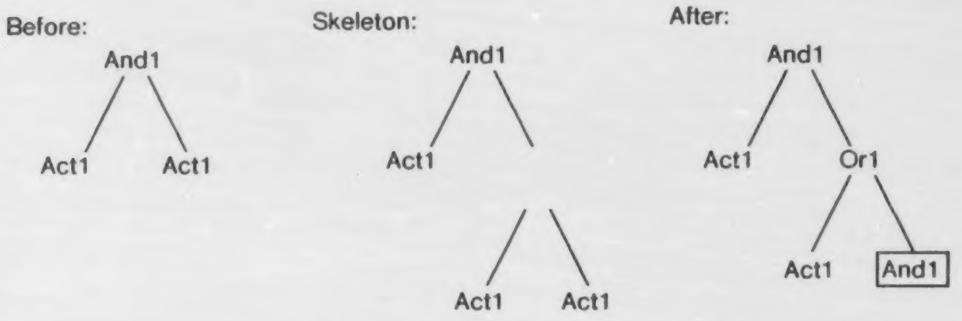


Figure 19–7
Fragments of an AOG before and after loop step schema has applied.
The boxed AND node indicates a recursive call.

Another step schema is the *loop schema*. It detects skeletons that are intended to introduce loops. Its recognition condition is somewhat more complicated. The skeleton's kid list must be two kids long. Both kids must be the same action, and moreover, that action must match a sister of the skeleton parent. Furthermore, the parent must be called from an AND that also has an instance of the goal. Figure 19–7 illustrates this. Essentially, the schema converts a procedure that can handle a single or double occurrence of `Act1` into a recursive procedure that handles arbitrarily long sequences of `Act1`'s. It does this when it detects a triple occurrence of `Act1`. The loop schema replaces one of the skeleton's kids with the right goal to create a tail recursion.

It is important to note that the loop schema can be configured differently. By leaving off part of its recognition condition, one can have it detect loops when only two occurrences of `Act1` are in the example. This predicts that a student who is shown two-column subtraction for the first time could infer the tail recursion needed for multicolumn subtraction. This prediction seems too strong to me, although I have no evidence against it. For what it's worth, one young student solemnly told a colleague, "It takes three to see a pattern. If there's only two, you don't have a pattern." The variability in the definition of the loop schema illustrates that step schemata can increase the tailorability of the theory. If the student had said, "It takes four to see a pattern," one could easily construct the corresponding loop schema.

Constraints on test patterns

More subtly, the meaning of some schemata, such as the preparatory step schema, entail some constraints on the patterns that are constructed to fill out the skeleton. The preparatory step schema seems intuitively to involve the notion of satisfying some precondition of the action being prepared for. This would entail that some precondition of the action ought to be a part of the test pattern of the new subprocedure. Thus, since T<B is a precondition of Diff, T<B ought to be part of the rule that pushes for borrowing.

In fact, step schemata *must* have such constraints on test patterns. If they don't, then they admit some empirically flawed skeletons. In fact, they admit all the skeletons that the lowest parent hypothesis rules out. Re-examination of figures 19–3 and 19–4 shows that both these skeletons meet the topological requirements of the preparatory step schema. In order to filter out the skeletons that lead to unobserved bugs, the preparatory step schema must either adopt the lowest parent hypothesis, blatantly violating Occam's razor, or it must use test patterns and preconditions in filtering the larger skeletons.

An experiment with step schemata

The three step schemata mentioned above were implemented, and a couple of subtraction curricula were run through Sierra. Although not the most extensive test in the world, this exercise indicated enough serious flaws in the step schema framework to warrant its rejection.

Before discussing those flaws, it is worth mentioning that this was not an expected or a welcome result. I had fully expected the step schemata to suffice for skeleton filtering. It seemed intuitively obvious at the time that skeleton acquisition ought to be constrained by the student's general knowledge of about procedures. Indeed, there were a number of talks where I sketched the notion of teleological rationalizations in glowing terms (step theory was given its name back in those days). There was a grand research programme waiting in the wings. The study of learning, as it occurs in current classes, was seen as paving the way for improved curricula: In the present, descriptive study, the step schemata of the naturally occurring teleology would be uncovered and formalized. Then the procedures and the curricula could be overhauled to conform to the natural teleology. However, the present study has uncovered no traces of a rationalization-based teleology. At this time, it appears that the data conform best to a simpler model, the topological one.

Step schemata block bugs

One problem with step schemata is that the learning they predict is too good to be true. When constructed to reflect intuitively salient teleology, they prevent the learner from acquiring several observed bugs. For instance, suppose the preparatory step schema constrains test patterns to check the preconditions of the action being prepared for. The rationalization for this is that if the precondition is satisfied, there is no need to prepare; if it's false, the newly acquired subprocedure should be executed. In subtraction, both borrowing and BFZ are instances of this preparatory step schema. If the precondition-checking teleology is built into the schema, then several borrowing and BFZ bugs cannot be generated by subprocedure acquisition (e.g., N–N-Causes-Borrow, Borrow-Treat-One-As-Zero). To regain the generation of these bugs, one would have to elaborate the schema's constraints or dispense with them altogether.

In a larger view, this flaw takes on added importance. Fixing precondition violations with preparatory steps has been a fixture in every study of teleology that I know of. If this rudimentary notion cannot be sustained undamaged by the data, then prospects look dim for teleological rationalizations as a whole.

Step schemata and tailorability

The worst problem with using step schemata is that the theorist must either fix the set of step schemata once and for all or live with a highly tailorable model. The tailorability makes the theory difficult to refute. As a case in point, I once thought there might be a schema for the notion of *preprocessing*. In particular, it seemed that borrowing could be construed as a preprocessing step for the multi-column traversal. Using this schema, the learner would acquire a procedure with two loops. The first loop would make all the columns easy by borrowing whenever necessary. The second loop would move across the columns again, taking the column difference. The preprocessing version of subtraction makes teleological sense, and generates a few unique bug predictions. However, none of these predictions have been verified by the data. Does this mean that the whole step schemata approach is wrong, or does it mean only that there is no preprocessing schema? There is no way to know without collecting more data. Not knowing whether the theory is really right is the price one sometimes pays for having a highly tailorable theory.

In short, the teleological bias will be rejected partly because one particular version of it does worse than the topological bias at predicting bugs and partly because it is too difficult to refute in general.

19.3 Summary

Test pattern induction is one of the most critical issues in the theory. Several bugs depend directly upon the biases used in their acquisition. Nonetheless, it has proved a very tricky issue to discover what those biases are. Two positions were considered. One was topological in character. Given a version space of all patterns that are consistent with the lesson's positive and negative instances, the topological bias has the learner keep a maximally general pattern as the test pattern. The other position is based on teleological rationalizations, cast as step schemata. These schemata act as filters on the possible generalizations of the examples offered by the otherwise unbiased inducer. The prototypical step schema is the preparatory step schema. It sanctions subprocedures that can be construed as preparing for some already existing step in the procedure.

The difference between the two kinds of bias are subtle despite the fact that the positions they represent view learning in radically different ways. The topological view is basically an empiricist viewpoint while the teleological viewpoint is strongly nativist. Nonetheless, the data does not argue strongly for one over the other. The issue is settled mostly on grounds of tailorability. Under one reasonable interpretation of the teleological bias, several bugs could not be generated. The bugs could not be generated because the teleology of the preparatory step schema matches the correct teleology for subtraction rather closely. Hence, it prefers correct subtraction procedures over the buggy ones. It cannot, therefore, generate certain bugs that the topological bias can generate. However, there is nothing sacred about the particular schemata that were used. If they were replaced by less stringent ones, then perhaps the bugs could be generated. This points out just how much control the theorist has over the predictions of the theory when step schemata are used. It is this tailorability that led to the downfall of the teleological approach.

The chapter discussed a second kind of learning that is very strongly associated with test pattern induction. It is the choice of skeleton for the new subprocedure. The skeleton expresses the attachment point of the new subprocedure to the procedure's current goal hierarchy. Essentially, skeleton induction is the control structure analog to test pattern induction. It decides "when" in the control environment to execute the new subprocedure; test pattern induction decides "when" in the external environment to execute it. The two biases just discussed, the topological bias and the teleological bias, extend to skeleton induction. Each becomes a little more complicated, since the topology of the control structure is a little more complicated than the topology of test patterns (i.e., trees versus sets). However, the same basic results are found for skeleton induction. Both biases work, but the topological bias is less tailorable.

Chapter 20
Summary: Bias Level

The bias level, chapters 17 to 19, uncovered and formalized the learner's biases concerning pattern induction and skeleton induction. It began by fulfilling a promise made earlier to show that test patterns and fetch patterns are actually distinct. This yielded the following hypothesis:

> *Two patterns*
> The representation uses different patterns for testing rule applicability and for focus shifting: Test patterns are used in choosing which OR rule to execute. Fetch patterns are used to shift the focus of attention (data flow).

This distinction was needed by the representation level in order to define the procedure representation language.

From the standpoint of learning, the representation language defines the form of patterns and subprocedure skeletons. Induction's job is to find all possible ways to fill in their forms in such a way that they are consistent with the examples. Because the representational hypotheses are so constraining, it is technically feasible to generate all these possibilities and check some of the resulting predictions. It was shown that pure, unbiased induction overgenerates. It acquires skeletons and patterns that human learners do not. Apparently, students have some biases.

Topological biases vs. teleological biases

The bias level contrasted two kinds of bias. *Topological biases* (which are ultimately shown to be the better biases) are based on maximizing or minimizing generality. They are "knowledge free" in that the biases can be computed directly from the topology of patterns and skeletons. *Teleological biases* postulate a knowledge base that contains teleological rationalizations. A teleological rationalization invents a plausible purpose for a new subprocedure. The bias is to accept only subprocedures that appear to have some purpose. For the sake of discussion, teleological rationalizations are represented as *step schemata* (Goldstein, 1974). The prototypical step schemata is the preparatory step schema. It sanctions subprocedures that can be construed as preparing for some already existing step. For instance, borrowing might be rationalized (correctly, in this case) as preparation for taking the column difference.

The evidence does not clearly favor topological biases over the teleological ones, except in the case of fetch patterns. For fetch patterns, the grammar serves as a repository for teleological rationalizations. Given that the grammar has, for instance, the teleological notion that boundary conditions are important, topological biases for fetch patterns yield high quality predictions. In the case of test patterns and skeletons, more than the grammar is needed to bias their induction. Teleological rationalizations would serve adequately, it seems. However, having a knowledge base of step schemata introduces a great deal of tailorability into the theory. Topological biases introduce no tailorability at all, yet they seem as good or better than the teleological biases in their empirical predictions. The remainder of this chapter discusses the topological biases.

The topological biases factor the bias issues along the lines of a subprocedure's components. There are four components of interest: (1) the test pattern on the adjoining rule, (2) the fetch patterns on the new AND's rules, (3) the subprocedure's parent OR, and (4) the subprocedure's kids.

The parent OR is the OR goal that the new subprocedure will be underneath. The kids are goals that the new subprocedure will call. The biases concerning these four components will be discussed in turn.

Topological biases for "when"

The test pattern and the parent OR are related in that both concern *when* the new subprocedure may be executed. The test pattern expresses external conditions. It functions as a predicate on problem states. The parent OR expresses internal conditions. It functions as a predicate on the top of the goal stack. Only if the parent OR goal is on the top of the stack may the procedure execute the new subprocedure. It is somewhat surprising that both aspects of "when" — the parent OR and the test pattern — are subject to the same topological bias. In both cases, the learner prefers maximizing *generality*. First test pattern bias will be discussed, then parent OR bias.

The range of test patterns that the model's inducer picks from is represented as a version space, ⟨S, G⟩, where G is the set of maximally general patterns and S is the set of maximally specific patterns. As examples are fed to the inducer, these two sets creep toward each other. The maximally general patterns in G become more specific. The maximally specific patterns in S become more general. Induction could be unbiased if it always happened that they came together (G=S). In this case, their contents would be the *only* patterns consistent with the examples, and bias would be superfluous. Given actual lessons, S and G never come close. A typical S pattern has a hundred relations; a typical G pattern has a half-dozen relations. The inducer may choose any pattern that is between the two sets G and S (technically, any pattern that is a subgraph of some $s \in S$ and a supergraph of some $g \in G$). The learner has on the order of 2^{100} possible patterns to choose among. The bias is to choose only patterns in G:

> *Maximally general test patterns*
> Given that ⟨S,G⟩ is the version space for a test pattern,
> `InduceTest` chooses an $g \in G$ as the test pattern.

Maximal generality is also the bias for choosing parent ORs. However, it is generality of a different kind. Skeleton induction first locates all possible parent ORs. That is, any of the parent OR's would lead ultimately to a new procedure that is consistent with all the examples. These parent ORs are all logically equivalent, in a sense. No choice of parent OR is more general than the others. However, choosing a parent OR that is low in the AOG (i.e., far from the root) causes future subprocedure acquisitions to create more general procedures. This is a new conception of generality, so it deserves a moment's discussion. Suppose one were adjoining disjunctions to And-Or propositions, starting with `(AND A B)`. If the inducer is shown `CB` as an example, the following logically equivalent expressions would both be consistent generalization of the example:

```
a. (AND (OR A C) B)          b. (OR (AND A B) (AND C B))
```

Expression *a* represents adjoining a subprocedure `C` to a low parent; *b* represents adjoining to a higher parent. When the inducer is shown `AD`, causing a second subprocedure `D` to be adjoined, the two expressions produce expressions that are not logically equivalent:

```
c. (AND (OR A C) (OR B D))    d. (OR (AND A (OR B D)) (AND C B))
```

Whereas *c* is true of all four of `{AB, AD, CB, CD}`, expression *d* is true of just the first three. Expression *c* has more generality, yet the cause of its generality is the initial attachment of `C`. The low attachment of `C` gave expression *a* greater *potential generality*. Given two logically equivalent expressions, one expression has more potential generality if there exists an induction algorithm and a sequence of examples that makes the first expression more general than the second.

The learner's bias for parent ORs is to chose the skeleton that maximizes potential generality. It chooses the lowest parent OR. The hypothesis says exactly this, albeit in a somewhat technical way:

> *Lowest parent*
> Given two subprocedures, A and B, for possible addition to a procedure P, if A is *lower* than B in that there is a path from P's root to A that passes through B, then `InduceSkeleton` chooses A.

Topological biases for "what"

We just saw that the biases for when to execute a subprocedure are based on maximizing generality. The other two components, fetch patterns and kids, are subject to the opposite bias. The learner chooses them in order to maximize *specificity*. The fetch pattern describes where the subprocedure's action takes place. The kid describes what that action is. Essentially, the two together establish *what* the subprocedure does. Viewed this way, it makes sense that they should have the same bias.

The range of choices for a fetch pattern is a version space, just as with test patterns. The learner's bias is to take a maximally specific pattern:

> *Maximally specific fetch patterns*
> Given that <S, G> is the version space for a fetch pattern,
> `InduceFetch` chooses any pattern in S as the fetch pattern.

However, unlike maximal generality, maximal specificity is essentially unbounded. Patterns can get infinitely large. If one pattern describes only a column and the other pattern describes the same column in the context of a problem, then the second pattern is more specific. A pattern that describes the same column and problem in the context of a page of problems would be an even more specific pattern. To put a limit on this boundless specificity, some maximal size is needed. Based on bug evidence, the following hypothesis seems to be the appropriate one:

> *Focus pattern*
> If <S, •> is the version space for a fetch pattern, and the fetch pattern is beneath goal G, and the fetch pattern will provide arguments to goal SG, a subgoal of G, then let I (for input) be the set of pattern variables corresponding to goal G's arguments and O (for output) be the set of all variables used for providing arguments to SG. The part-whole tree of a pattern in S is the minimal tree necessary to span the set I∪O. Variables outside this part-whole tree are dropped from the pattern, along with all relations that mention them.

Essentially, the fetch patterns are chosen to be the maximally specific patterns that might prove useful in disambiguating fetches. The focus pattern hypothesis asserts that parts of the part-whole tree that the input-output variables do not reside in are too distant to be useful in disambiguating various ways to match the output variables.

The other half of the "what" bias concerns choosing kids for a skeleton subprocedure. As with choosing a parent, unbiased induction leaves several choices open. Empirical evidence motivates the following biases:

Fewest kids
If two subprocedures, A and B, have the same parent OR, and A has fewer kids than B, then `InduceSkeleton` chooses A.

Lowest kids
If two subprocedures, A and B, have the same parent OR and the same number of kids, and each of A's kids is lower than or equal to the corresponding kid in B, then `InduceSkeleton` chooses A, where "lower" is defined as in the lowest parent hypothesis.

The lowest kids hypothesis is an instance of the bias toward maximal specificity. As with the parent OR choice, it is potential specificity rather than current specificity that is being maximized. The lowest kids hypothesis maximizes potential specificity. The other hypothesis, fewest kids, is a bias toward generality. Surprisingly, it has precedence over the bias towards specificity, the lowest kids hypothesis. The data clearly require that it have precedence — it is impossible to learn tail recursive loops if the precedence is reversed. Why this is, is still a mystery.

The learners' topological biases fall into a coherent pattern. The following table illustrates it:

	When Max. generality	What Max. specificity
Skeleton	`Parent OR`	`Kids`
Patterns	`Test pattern`	`Fetch patterns`

The biases for when to execute the new subprocedure are in favor of maximizing generality. The learners prefer to use the new subprocedure as much as possible. They choose a parent OR and a test pattern that will cause maximal usage of the new subprocedure. This is echoed in the conflict resolution principle that stipulates that the most recently acquired rule is preferred whenever more than one rule is applicable and has a true test pattern (see section 10.5). So, all principles fit into a picture of students who tend to exercise their new subprocedures as much as possible.

The biases concerning *what* the subprocedure should do go the opposite direction. In the case of fetch patterns, there is a clear preference for maximally specific patterns. In the case of the skeleton's kids, there is a somewhat mixed preference for maximizing potential specificity.

Matching

The bias toward maximal specificity of fetch patterns makes sense. The basic job that a fetch pattern does is disambiguate which of the many visible objects in the problem state the procedure should shift its focus to. The best way to do that is to remember everything about the exemplary fetches that might be even remotely useful (i.e., choose a maximally specific fetch pattern). During problem solving, this highly specific description will often not apply exactly, but that is okay: the procedure takes the closest match to the fetch pattern. That is, solver views the current problem state in such a way that it approximates the lesson situation as closely as possible. Thus, it minimizes the risk of fetching the wrong objects. This way of using fetch patterns is captured in two hypotheses:

> *Fetch pattern matching*
> Let b be a binding of the fetch pattern's variables to objects of the current problem state such that all input variables have their correct bindings. Let P_b be the subset of relations in fetch pattern P that are true (satisfied) by b. Then b is a valid match for the fetch pattern only if (1) there is no b' such that $P_b \subset P_{b'}$ and $P_b \neq P_{b'}$, and (2) all the part-whole relations of P are in P_b.
>
> *Ambiguity impasse*
> If a fetch pattern matches ambiguously so that the output variables have two or more distinct bindings, then an ambiguity impasse is triggered.

The second hypothesis reflects the view that disambiguation is the main job of a fetch pattern. If it fails, then local problem solving will have to take over.

Test patterns, on the other hand, clearly have a different role. They must report true or false. The only way to achieve this function (and still generate certain critical bugs, the fetch bugs) is to use exact matching for test patterns:

> *Test pattern match*
> A test pattern is considered to be true if and only if it matches exactly (i.e., all its relations are true in the current problem state).

Function biases

The representation level presented a nearly complete formalization of the learner and the solver. It left six functions undefined:

```
SkeletonFilter
InduceTest
InduceFetch
InduceFunctions
Test
Fetch
```

The hypotheses developed in bias level defined all of these functions except for `InduceFunctions`. That bias has not been discussed yet. The evidence is so clear and intuitively compelling that it is unnecessary to devote a whole chapter to it. The basic idea can be illustrated with the usual example, learning the BFZ lesson. If the core procedure does not have `REGROUP` in it (which occurs when the procedure is learned from a lesson sequence that does not have a special regrouping lesson), then there are three kids for the new subprocedure: (1) decrementing the hundreds column, (2) adding ten to the tens column, and (3) decrementing the tens column to nine. Each of these kids is a call to the `OVRWRT` goal, passing it a location and a number. To generate the number, each of the three kids has a function nest. The following are the choices for each nest that are inductively valid:

Decrementing the hundreds:

```
1. (Sub (Read ARG1) (One))
2. (Sub1 (Read ARG1))
```

Adding ten to the tens:

```
1. (Concat (One)(Zero))
2. (Concat (One)(Read NV1))
3. (QUOTE 10)
```

Decrementing the tens column

```
1. (Sub (Read NV1)(One))
2. (Sub (Read NV1)(Read R47))
3. (Sub1 (Read NV1))
4. (QUOTE 9)
```

The variables, `ARG1`, `NV1` and `R47`, come from various fetch patterns. It doesn't matter what they mean. The point is only that each nest is inductively valid in that they are consistent with all possible examples. Only a bias of some kind can be used to eliminate them. It can be shown that only the last nest of each of the sets of nests above is empirically correct.

For the first kid, the choice is between `Sub` and `Sub1`. Suppose the inducer chooses `Sub`. This means that a precondition violation will occur whenever `ARG1` is zero, as it will be when the procedure is applied to solve problems that require borrowing across multiple zeros. In this case, one of the observed repairs is Backup, which, as in the bug Smaller-From-Larger-Instead-of-Borrow-From-Zero (see sections 9.1 or A9.1), causes a secondary impasse and further local problem solving. The secondary impasse also involves `Sub`, but in the context of answering a column. It is the familiar T<B impasse. This secondary impasse is solved by a different repair than Backup. So the same precondition violation, trying to `Sub` a larger number from a smaller number, is repaired two different ways. The observed bug is stable. It does this dual repair consistently. Although the theory allows such flipping back and forth between repairs, it is clear that the stability of the bug is better modelled if the decrement-zero impasse is a different impasse than the T<B impasse. This falls out naturally if `Sub1` is chosen instead of `Sub`. This means that the decrement-zero impasse will be a violation of `Sub1`'s precondition, and the T<B impasse will be a violation of `Sub`'s precondition. The two impasses are formally distinct. A different patch for each is quite plausible, and cleanly accounts for the observed bug. Apparently, learners are biased to chose `Sub1` over `Sub` in the case of the first subprocedure kid.

In the case of the third subprocedure kid, the evidence is stronger but more complicated. The basic finding is that when the rule deletion operator removes the second rule of the new subprocedure, the third rule is called to decrement a zero, which would normally be a ten. This would generate an impasse. However, no bugs corresponding to this impasse have been found. Consequently, none of the `Sub` or `Sub1` nests are in use, since they would all generate impasses. The last nest, which is just the constant 9, is evidently the nest that learners prefer.

These facts argue that the biases of learners with respect to function induction is simply to chose the function nest with the fewest argument places, where a constant counts as having no argument places. This is captured in the following hypothesis:

> *Smallest arity*
> If two subprocedures, A and B, are identical except for a function nest, and the arity of A's nest is smaller than the arity of B's nest, then `InduceFunctions` prefers A, where the arity of a function nest is the sum of the number of argument places in its functions (i.e., constants and nullary functions count 0, unary functions count 1, binary functions count 2, etc.).

This is a rather minor bias that has a clear intuitive interpretation. Suppose that executing unary facts functions requires less use of cognitive resources than executing a binary facts function, and that retrieving a constant is even easier. The smallest arity bias then means that students prefer function nests which reduce their cognitive load during execution. So this hypothesis is rather plausible in addition to having a degree of empirical support.

Chapter 21
Conclusions

Danny Bobrow once said that AI researchers tend to stand on the toes of their predecessors rather than on their shoulders (Bobrow, 1973). I have tried to stand on a few shoulders. In particular, the argumentative methodology comes from Chomskyan linguistics. The induction technology comes from Winston and his many successors. The representational framework is heavily influenced by Newell and Simon's work on production systems. The basic notion of local problem solving orginated with John Seely Brown. Just as I have built on the work of others, I would like to think that this theory provides something worth building on. This chapter gives a somewhat personal assessment of the strengths and weaknesses of the theory, and suggests some directions for future research.

21.1 Strengths and weaknesses in the theory

The architectural level is solid. One of its basic notions is that procedures are interpreted with the aid of a local problem solver. The existence of local problem solving is indisputable. A great deal of bug data and especially bug migration data supports the existence of the local problem solver. Exactly how it works, i.e., the particular set of repairs and impasses, is not perfectly understood. There is no performance model for the local problem solver. Nonetheless, the basic notions of local problem solving seem likely to survive a more detailed investigation.

The other fundamental idea of the architecture level is felicity conditions. They are an analysis of what makes lessons simpler to learn from than random examples. As far as I know, this theory is the first to address the question of why lessons help the learner learn. Having asked the question, the answers are fairly obvious. Only the show-work principle was a surprise. It was a surprise, I suppose, because few AI researchers have looked at the problem of inducing function compositions. Although inducing disjunction is a well-known problem, inducing nests of functions has received little attention. Having uncovered the problem, the felicity condition that solves it is quite apparent. Are there undiscovered felicity conditions? Since Sierra is, in fact, able to induce procedures, I doubt that there are major undiscovered induction problems. The disjunction problem and the invisible objects problems seem to be the only "insoluble" induction problems. Although the architecture level's solutions to these problems, which rely heavily on the lesson boundaries, might actually turn out to be slightly inaccurate, the fundamental induction problems can be expected to be permanent features of the theory.

The representational level is somewhat problematical. Given the standard notion of control flow, data flow and interface, it seems a reasonable analysis of the structure of human procedures. However, recent work by Brian Smith (1982) indicates that these fixtures of computer science may not be the only way to understand computation. It is possible that a much better version of the representation level will be discovered when the new ideas of computation become better understood. Ideas about computation serve as a set of distinctions for analyzing human procedural knowledge. Using familiar ideas about computation, one standard distinction was found to be irrelevant. The applicative hypothesis essentially erases the distinction between control flow and data flow. It says that the two kinds of information move together. Perhaps there are other distinctions that, while not familiar ones now, may add to the clarity of the analysis of procedural knowledge. In particular, the issue of whether focus of attention is intensional or extensional (which is discussed in appendix 8) seems to be a place where critical distinctions are needed and lacking.

As part of a theory of learning and local problem solving, the representation language functions merely as a set of absolute constraints (as opposed to binary or relative constraints, such as the learner's biases for maximal generality). The representation defines a format for subprocedures, and the learner just fills in the blanks. The representation defines a runtime state, and the local problem solver just manipulates it in simple ways. If we put aside all claims that the language represents the structure of mentalese, then the veracity of the language lies in whether or not it correctly draws the boundaries between learnable and unlearnable subprocedures, and between occurring and non-occurring local problem solving. In both learning and local problem solving, the assessment of boundaries is complicated by the fact that there are more constraints on the model than just the representational ones. In learning, the bias constraints filter out most of the subprocedures that the representation would allow the learner to output. In local problem solving, only stipulated repairs and impasses are used; every possible change to the runtime state is not a sanctioned repair. Despite these complications, the boundaries seem quite well set by the representation language.

The only place where there is some uncertainty concerns loops. As the lesson traversal of section 2.8 shows, a large number of unobserved bugs are generated in the course of learning the main column loop. Most of these could be avoided if (1) the representation had some special "Foreach" loop construction for processing written lists, such as the list of subtraction columns, and (2) the learner was biased to choose it instead of the tail recursive formulation of loops whenever both were possible. Unfortunately, the data concerning bugs in the early lessons of subtraction is quite sparse (those lessons occur in second grade; the youngest students in the subtraction studies were in third grade), and the later stages of subtraction do not apparently provide an opportunity to use this "Foreach loop" construction. The existence of the Foreach loop construction seems certain, but the details of its formulaiton have been left undecided until more data is available. Modulo this issue, the representation level seems quite solid.

The bias level is where the greatest uncertainty lurks. Only the system of acquiring and matching fetch patterns is well supported. The grammar provides most of the constraints here; the fetch bugs vouch that it provides the right ones. However, the biases for inducing skeletal subprocedures and test patterns are not very well supported. Teleological rationalizations provide a viable alternative that may be able to generate many of the bugs that the topological approach does not generate. However, merely providing a subject parameter filled by a set of step schemata is an intolerable retreat from explanatory adequacy. A way to reduce tailorability to acceptable levels might be to provide a constricting representation language for step schemata. Just as grammars embed the universal aspects of students' notational knowledge, there might be a grammatical way to embody universal teleological notions. Such universal notions would have to be somewhat domain specific in order to have enough strength to explain the existence of the various observed teleological rationalizations. I expect that notions of compensation and symmetry would be important for written calculations but that cause and effect might be unimportant.

To summarize: The architectural level seems quite solid. The representation level serves well as a source constraint on learning and local problem solving, but its deeper significance, especially its relationship to mentalese, is not yet clear. The bias level seems incomplete. Adding a mini-theory of a teleological rationalizations may improve it.

21.2 Directions for future research

There are many directions for further research. Some were just mentioned. Some others that were mentioned in earlier chapters are:

1. The theory should be applied to other tasks, such as algebra equation solving.
2. Optimization lessons should be analyzed and incorporated in the theory.
3. Critics and their acquisition are important issues that need immediate attention.
4. Grammar acquisition is an issue that seems ripe for studying.
5. The relationship between slips and deletion needs investigation. It is likely that a rule deletion is just a well-practiced slip.

In terms of applying the theory to education, perhaps the most important and difficult area of future research concerns the long-term retention of procedures. People remember much more than core procedures. Since some repair-generated bugs are stable, people must be remembering patches, the association of a repair with an impasse. One aspect of retention is of key interest to educators: how much drill is needed? Should it be lumped or distributed? Currently, textbooks use about 50 hours of distributed drill to teach subtraction. Most of that time is spent on review lessons. Why? Neves' analysis of algebra review lessons (Neves, 1981; see section 4.3) seems to say that students don't remember a complete procedure between lessons, but instead remember a set of induction heuristics that enable them to reconstruct the procedure given the brief examples of a review lesson. They don't remember the procedure per se, they remember how to learn it.

What little data there is on long-term bug stability paints a confusing picture. One mystery is reversion: the student reverts to using a bug that appeared to have been remediated.

The practical importance of studying procedure memory is that it would make this theory a valuable tool for curriculum design. Currently, the model can be used to establish a minimal content for lessons. It determines whether or not a lesson is missing certain kinds of examples and exercises. Given some curricular objective for the lesson, it establishes a "requisite variety" for a lesson's example. Although it establishes a lower bound on content, textbook publishers need to know an upper bound and an average, as well. They need to know how many lessons to budget for a certain objective. Since the present theory cannot tell them this, it is, at best, half a tool. But it is still better than no tool at all.

Not only does the theory critique individual lessons, suggesting examples that should be added, it can critique the lesson sequence as a whole. Some textbooks leave out critical lessons (one leaves out the borrow-from-zero lesson!) while retaining ones that are unnecessary (according to the present theory, which doesn't address memory-related issues).

It could be argued that these applications of the theory will be short lived. As calculators become ubiquitous, there may be less need to teach students efficient arithmetic skills. Instead, the procedure might be used to introduce concepts more useful in today's society — concepts such as procedures, data structures (e.g., base-10 notation), design (teleological semantics) and debugging. "Rote" learning of the surface structure of arithmetic procedures may disappear. Nonetheless, there will always be many procedures that adults learn by rote, and someone has to design the lessons that teach those procedures. Almost every piece of computer software sold today is accompanied by a manual which teaches the user how to use it. Whenever a company markets a new machine, its service personal must be trained to repair it. Whenever a bank creates a new kind of account, its personnel must be taught new procedures for opening, maintaining and closing the account. In most cases, the people learning these procedures are not interested in the deep, teleological structure that underlies the procedures' design; they just want to know what steps to follow. "Rote" learning is all they want and perhaps it is all that they need in order to work efficiently. If this theory can be generalized to tasks outside the domain of written symbol manipulation, it may be a tool for the training industry to use in rapidly designing curricula to meet the needs of its clients and their students.

References

Anderson, J.R. Acquisition of cognitive skill. *Psychological Review, 89,* 369–406, 1982.

Anderson, J.R. *Cognitive Psychology and its Implications.* San Francisco, CA: Freeman, 1980b.

Anderson, J.R., Greeno, J., Kline, P.J., & Neves, D.M. Acquisition of problem solving skill. In J.R. Anderson (Ed.) *Cognitive skills and their acquisition.* Hillsdale, NJ: Erlbaum, 1981.

Anderson, J.R., Kline, P.J. & Beasley, C.M. A general learning theory and its application to schema abstraction. *The psychology of learning and motivation,* 1979, *13,* 277-318.

Anderson, J.R., Kline, P.J. & Beasley, C.M. Complex learning processes. In R.E. Snow, P. A. Federico, & W.E. Montague (Eds.), *Aptitude, learning, and instruction: Cognitive process analyses.* Hillsdale, NJ:Erlbaum, 1980.

Anzai, Y. & Simon, H.A. The theory of learning by doing. *Psychological Review,* 1979, *86,* 124-140.

Ashlock, R.B. *Error patterns in computation.* Columbus, OH: Bell and Howell, 1976.

Austin, J.L. *How to do things with words.* New York: Oxford University Press, 1962.

Badre, N.A. *Computer learning from English text* (memo ERL-M372). Berkeley, CA: University of California at Berkeley, Electronic Research Laboratory, 1972.

Berwick, R. Cognitive efficiency, computational complexity and the evaluation of grammatical theories. *Linguistic Inquiry,* 1982, *13* (2).

Biermann, A.W. & Feldman, J.A. A survey of results in grammatical inference. In S. Watanabe (ed.), *Frontiers of Pattern Recognition.* New York: Academic Press, 1972.

Bobrow, D.G. Address given at Third International Joint Conference on Artificial Intelligence, Stanford CA, 1973.

Brown, J.S. A symbiotic theory formation system. Irvine, CA: University of California at Irvine, Dept. of Computer Science technical report 17, 1972.

Brown, J.S. Steps toward automatic theory formation. *Proceedings Third International Joint Conference on Artificial Intelligence,* 1973.

Brown, J.S. & Burton, R.B. Diagnostic models for procedural bugs in basic mathematical skills. *Cognitive Science,* 1978, *2,* 155-192.

Brown, J.S. & VanLehn, K. Repair Theory: A generative theory of bugs in procedural skills. *Cognitive Science,* 1980, *4,* 379-426.

Brown, R. *Use of analogy to achieve new expertise.* (MIT-AI-TR-403). Cambridge, MA: Massachusetts Institute of Technology, Artificial Intelligence Laboratory technical report, 1977.

Brownell, W.A. The evaluation of learning in arithmetic. In *Arithmetic in general education*. 16th Yearbook of the National Council of Teachers of Mathematics. Washingtion, DC: N.C.T.M., 1941.

Brueckner, L.J. *Diagnostic and remedial teaching in arithmetic*. Philadelphia, PA: John C. Winston, 1930.

Bunderson, C.V. *Cognitive bugs and arithmetic skills: Their diagnosis and remediation* (interim tech. rep.). Provo, Utah:Wicat, April 1981.

Burton, R.B. DEBUGGY: Diagnosis of errors in basic mathematical skills. In D. H. Sleeman & J. S. Brown (eds.), *Intelligent tutoring systems*. London: Academic Press, 1981.

Buswell, G.T. *Diagnostic studies in arithmetic*. Chicago, IL: University of Chicago Press, 1926.

Carbonell, J.G. Learning by analogy: Formulating and generalizing plans from past experience. In R.S. Michalski, J.G. Carbonell, & T.M. Mitchel (eds.), *Machine Learning, An artifical intelligence approach*, Palo Alto, CA: Tioga Press, 1983a.

Carbonell, J.G. Derivational analogy in problem solving and knowledge acquisition. In R.S. Michalski (ed.) *Proceedings of International Machine Learning Workshop*, Urbana, IL: University of Illinois, 1983b.

Carry, L.R., Lewis, C. & Bernard, J.E. Psychology of equation solving: An information processing study. (Technical report.) Austin, TX: University of Texas at Austin, Department of Curriculum and Instruction, 1978.

Chase, W.G. & Ericsson, K.A. Skill and working memory. In G.W. Bower (ed.), *The Psychology of Learning and Motivation*, Vol. 16, New York: Academic Press, in press. Currently available as Tech. Rep. ONR-6, Pittsburgh, PA: Carnegie-Mellon University, Dept. of Psychology, 1982.

Chomsky, N. *The logical structure of linguistic theory*. New York: Plenum Press, 1975.

Cohen, P.R. & Perrault, C.R. Elements of a plan-based thoery of speech acts. *Cognitive Science, 3*, 1979, 177–212.

Cohen, P.R. & Feigenbaum, E.A. *The Handbook of Artificial Intelligence*. Los Altos, CA: William Kaufmann, 1983.

Cox, L.S. Diagnosing and remediating systematic errors in addition and subtraction computations. *The Arithmetic Teacher*, 1975, *22*, 151–157.

Dansereau, D.F. An information processing model of mental multiplication. Pittsburgh, PA: Carnegie-Mellon University, Dept. of Psychology, unpublished PhD dissertation, 1969.

de Kleer, J. Causal and teleological reasoning in circuit recognition (AI-TR-529). Cambridge, MA: Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 1979.

de Kleer, J. & Brown, J.S. Mental models of physical mechanisms and their acquisition. In J.R. Anderson (ed.), *Cognitive skills and their acquisition*. Hillsdale, NJ: Erlbaum, 1981.

Dennis, J.B. First version of a data flow procedure language. *Proc. of Symposium on Programming*, Institut de Programmation, U. of Paris, April 1974, pp. 241-271.

Denofsky, M.E. How near is near? (AIM-344). Cambridge, MA: Massachusetts Institute of Technology, Artificial Intelligence Laboratory technical report, 1976.

Dietterich, T.G. & Michalski, R.S. Inductive learning of structural descriptions: Evaluation criteria and comparative review of selected methods. *Artificial Intelligence*, 1981, *16*, 257-294.

Ernst, G.W. & Newell, A. *GPS: A case study in generality and problem solving.* New York: Academic Press; 1969.

Fahlman, S.E. A planning system for robot construction tasks (AI-TR-283). Cambridge, MA: Massachusetts Institute of technology, Artificial Intelligence laboratory technical report, 1973.

Feldman, J. Some decidability results on grammatical inference and complexity. *Information and Control*, 1972, *20*, 244–262.

Fikes, R.E., Hart, P.E. & Nilsson, N.J. Learning and executing generalized robot plans. *Artificial Intelligence*, 1972, *3*, 251-288.

Fitts, P.M. & Posner, M.I. *Human Performance.* Belmont, CA: Brooks/Cole, 1967.

Floyd, R.W. Syntactic analysis and operator precedence. *Journal of the ACM*, 1963, *10*, 316-333.

Fodor, J.A. *The language of thought.* New York: Crowell, 1975.

Fodor, J.A. *Representations: Philosophical essays on the foundations of cognitive science*, Cambridge, MA: MIT Press, 1981.

Fodor, J.A. *The Modularity of Mind*, Cambridge, MA: Bradford Press, 1983.

Forgy, C. & McDermott, J. The OPS2 reference manual. Pittsburgh, PA: Carnegie-Mellon University Department of Computer Science memo, 1978.

Friend, J. & Burton, R.B. Teachers guide for DEBUGGY results. Palo Alto, CA: Xerox Palo Alto Research Center, CIS working paper, 1980.

Fu, K. & Booth, T. Grammatical inference: Introduction and survey, *IEEE Transactions on System, Man, and Cybernetics*, 1975, *5*, 95–111.

Gelman, R. & Gallistel, C.R. *The child's understanding of number.* Cambridge, Ma.: Harvard University Press, 1978.

Green, C. & Barstow, D. Some rules for the automatic synthesis of programs. *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, 1975.

Greeno, J.G. Riley, M.S. & Gelman, R. Conceptual competence and young children's counting. Unpublished manuscript, forthcoming.

Grice, H.P. Logic and conversation. In D. Davidson & G. Harmon (eds), *Semantics of Natural Language.* Dordrecht: Reidel Press, 1975..

Gold, E.M. Language identification in the limit. *Information and Control*, 1967, *10*, 447–474.

Goldstein, I.P. Understanding simple picture programs. (Rep. No. AI-TR-294). Cambridge, MA:Massachusetts Institute of Technology, Artificial Intelligence Laboratory, technical report, 1974.

Goldstein, I.P. & Bobrow, D.B. Descriptions for a programming environment. *Proceedings of the first annual conference of the National Association for Artificial Intelligence*, Stanford, CA, 1980, 187–194.

Goodman, N. *Fact, Fiction and Forecast.* New York: Bobbs-Merrill, 1955.

Gordon, D. & Lakoff, G. Conversational postulates. In D. Adams, M.A. Campbell, V. Cohen, J. Lovins, E. Maxwell, C. Nygren, & J. Reighard (Eds.) *Papers from the seventh regional meeting of the Chicago Linguistic Society.* Chicago: University of Chicago Department of Linguistics, 1971, 63-84.

Haviland, S.E. Buggy's analysis of the TORQUE Wellesley Data. Palo Alto, CA: Xerox Palo Alto Research Center working paper, 1979.

Hayes-Roth, B. & Hayes-Roth, F. Concept learning and the recognition and classification of exemplars. *Journal of Verbal Learning and Verbal Behavior*, 1977, *16*, 321–338.

Hayes-Roth, F. & McDermott, J. Learning structured patterns from examples. *Proceedings of the Third International Joint Conference on Pattern Recognition,* Stanford, CA, 1976, 419–423.

Hayes-Roth, F. & McDermott, J. An interference matching technique for inducing abstractions. *Communications of the ACM*, 1978, *21*, 401-411.

Hempel, C.G. Studies in the logic of confirmation, *Mind*, 1945, *54*, 1–26, 97-121.

Henderson, P. & Morris, J. A lazy evaluator. SIGPLAN-SIGACT Symposium on principles of programming languages, Atlanta, GA, pp. 95-103, 1976.

Horning, J.J. A study of grammatical inference (Rep. No. CS-130) Stanford, CA: Stanford University, Computer Science Dept., 1969.

Iba, G.A. Learning disjunctive concepts from examples. Cambridge, MA: Massachusetts Institute of Technology, Artificial Intelligence laboratory technical report 548, 1979.

Kaplan, R.M. A competence based theory of psycholinguistic performance. Psychology colloquium at Stanford University, Stanford, CA, May, 1981.

Keil, F.C. Constraints on knowledge and cognitive development. *Psychological Review*, 1981, *88*, 197-227.

Klahr, D. & Robinson, M. Formal assessment of problem-solving and planning processes in preschool children. *Cognitive Psychology*, 1981, *13*, 113-148.

Langley, P. Rediscovering physics with Bacon.3. In *Proceedings of the Sixth International Joint Conference on Artificial Intelligence.* Tokyo, Japan, 1979.

Langley, P. Finding common paths as a learning mechanism. *Proceedings of the Third National Conference of the Canadian Society for computational Studies of Intelligence*, 12-18, 1980.

Langley, P. A general theory of discrimination learning. Pittsburgh, PA: Carnegie-Mellon University, manuscript submitted for publication.

Lankford, F.G. *Some computational strategies of seventh grade pupils* (ERIC document). Charlottesville, VA: University of Virginia, 1972.

Lawler, R. The progressive construction of mind. *Cognitive Science*, 1981, *5*, 1-30.

Lenat, D.B. & Brown, J.S. Why AM and Eurisko appear to work. *Proceedings of the 1983 National Conferences on Artificial Intelligence*, Los Altos, CA: Kaufman, 1983.

Lewis, C. Production system models of practice effects. Ann Arbor, MI: University of Michigan, unpublished doctoral dissertation, 1978.

Manna, Z. & Waldinger, R. A deductive approach to program synthesis (Rep. No. 78-690). Stanford, CA: Stanford University, Computer Science Dept. technical report, 1978.

McDermott, J. & Forgy, C.L. Production system conflict resolution strategies. In D. A. Waterman & F. Hayes-Roth (eds.). *Pattern-directed inference systems.* New York: Academic Press, 1978.

McDonald, F.J. & Elias, P.J. The effects of teaching performances on pupil learning. Princeton, NJ: Educational Testing Service, final project report, 1975.

Michalski, R.S. On the quasi-minimal solution of the general covering problem. *Proceedings of the Fifth International Federation on Automatic Control*, 1969, *27*, 109-129.

Michalski, R.S. Variable-valued logic and its applications to pattern recognition and machine learning. In D.C. Rine (Ed.), *Computer science and multiple-valued logic theory and applications.* Amsterdam: North-Holland, 1975.

Mitchell, T.M. Generalization as search. *Artificial Intelligence*, 1982, *18*, 203-226.

Mitchell, T.M., Utgoff, P.E., & Banerji, R.B. Learning problem-solving heuristics by experimentation. In R.S. Michalski, T.M. Mitchell, & J. Carbonell, (Eds.), *Machine Learning.* Palo Alto, CA: Tioga, 1983.

Mitchell, T.M., Utgoff, P.E., Nudel, B., & Banerji, R.B. Learning problem-solving heuristics through practice. *IJCAI 7*, 127-134, 1981.

Mostow, D. J. *Mechanical transformation of task heuristics into operational procedures.* (Rep. No. CS-81-113) Pittsburgh, PA: Carnegie-Mellon University, Computer Science Dept., 1981.

Neches, R. Promoting self-discovery of improved strategies. Paper presented at the annual conference of the American Educational Research Association, San Francisco, CA, 1979.

Neches, R. HPM: A computational formalism for heuristic procedure modification. *Proceedings of Seventh International Joint Conference on Artificial Intelligence*, 1981.

Neves, D.M. *Learning procedures from examples*, Unpublished doctoral dissertation, Department of Psychology, Carnegie-Mellon University, Pittsburgh, PA, 1981.

Neves, D.M. & Anderson, J.R. Knowledge compilation: Mechanisms for the automatization of cognitive skills (Rep. No. 80-4). Pittsburgh, PA: Carnegie-Mellon University, Department of Psychology, 1981.

Newell, A. & Rosenbloom, P.S. Mechanisms of skill acquisition and the law of practice. In J. Anderson (ed.) *Cognitive Skills and their Acquisition.* Hillsdale, N.J.: Lawrence Erlbaum Associates, 1981.

Newell A. & Simon H.A., *Human Problem Solving*. Englewood Cliffs, NJ: Prentice Hall, 1972.

Norman, D.A. Categorization of Action Slips. *Psychological Review*, 1981, *88*, 1–15.

Pinker, S. Formal models of language learning. *Cognition*, *7*, 217–283, 1979.

Purcell, S.C. Understanding hand-printed algebra for computer tutoring (AIM-445). Cambridge, MA: Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 1977.

Pylyshyn, Z.W. Computation and cognition: issues in the foundations of cognitive science. *The Behavioral and Brain Sciences*, 1980.

Resnick, L. Syntax and semantics in learning to subtract. In *Addition and Subtraction: A cognitive perspective*. T. Carpenter, J. Moser, & T. Romberg (Eds.) Hillsdale, NJ: Erlbaum, 1982.

Rich, C. Inspection methods in programming (AI-TR-604). Cambridge, MA: Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 1981.

Rich, C. & Shrobe, H.E. Initial report on the Lisp programmer's apprentice (AI-TR-354) Cambridge, MA: Massachusetts Institute of Technology, Artificial Intelligence laboratory, technical report, 1976.

Roach, E. & Mervis, C.B. Family resemblances: Studies in the internal structure of categories. *Cognitive Psychology*, *7*, 573–605, 1975.

Roberts, G.H. The failure strategies of third grade arithmetic pupils. *The Arithmetic Teacher*, 1968, *15*, 442–446.

Sacerdoti, E. *A Structure for Plans and Behavior*. New York: Elsevier North-Holland, 1977.

Samuel, A.L. Some studies in machine learning using the game of checkers. In E.A. Feigenbaum & J.A. Feldman (Eds.). *Computers and thought*. New York: McGraw-Hill, 1963.

Searle, B., Friend, J., & Suppes, P. *The radio mathematics project: Nicaragua 1974-1975*. Stanford, CA: Institute for Mathematical Studies in the Social Sciences, Stanford University, 1976.

Searle, J.R. *Speech acts: An essay in the philosophy of language*. Cambridge: Cambridge University Press, 1969.

Simon, H.A. *The Sciences of the Artificial*. Second edition. Cambridge, MA: MIT Press, 1981.

Simon, A. & Goldberg, E. (eds.) *Mirrors for Behavior III: An anthology of observation instruments*. Wyncote, PA: Communication Materials Center, 1974.

Shaw, D.J., Standiford, S.N., Klein, M.F. & Tatsuoka, K.K. Error analysis of fraction arithmetic — selected case studies. (Rep. 82-2-NIE), Urbana, IL: University of Illinois, Computer-based Education Research Laboratory, 1982.

Sleeman, D.H. Basic algebra revisted: a study with 14 year olds. Manuscript in preparation, forthcoming.

Sleeman, D.H. An attempt to understand pupil's understanding of basic algebra. Submitted for publication.

Smith, B.C. Reflection and semantics in a procedural language (LCS-TR-272). Cambridge, MA: Massachusetts Institute of Technology, Laboratory for Computer Science, 1982.

Stefik, M.J. Planning with constraints (Rep. No. 80-784) Stanford, CA: Stanford University, Computer Science Dept., 1980.

Stoy, J.E. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory,* Cambridge, MA: MIT Press, 1977.

Suchman, L. Office procedures as practical action (CIS-32). Palo Alto, CA: Xerox Palo Alto Research Center, 1980.

Sussman, G.J. *A computational model of skill acquisition.* New York: Springer Verlag, 1976.

Sussman, G.J., Winograd, T. & Charniak, E Micro-planner reference manual (AIM-302A). Cambridge, MA: Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 1971.

Tatsuoka, K.K. & Baillie, R. Rule space, the product space of two score components in signed-number subtraction: an approach to dealing with inconsistent use of erroneous rules. (Rep., No. 82-3-ONR) Urbana, Ill.: University of Illinois, Computer-based Education Research Laboratory, 1982.

Tversky, A. Features of similarity. *Psychological Review, 84*, 327–351, 1977.

VanLehn, K. *Bugs are not enough: Empirical studies of bugs, impasses and repairs in procedural skills* (Tech. Rep. CIS-11). Palo Alto, Calif.: Xerox Palo Alto Research Center, 1981.

VanLehn, K. On the representation of procedures in Repair Theory. In H. Ginsberg (ed.) *The development of mathematical thinking.* New York: Academic Press, 1983.

VanLehn, K. & Brown, J.S. Planning Nets: A representation for formalizing analogies and semantic models of procedural skills. In R.E. Snow, P.A. Federico, & W.E. Montague (Eds.), *Aptitude, learning and instruction: Cognitive process analyses.* Hillsdale, NJ: Erlbaum, 1980.

VanLehn, K., Brown, J.S. & Greeno, J.G. Competitive argumentation in computational theories of cognition. In W. Kinsch, J. Miller & P. Polson (Eds.) *Methods and Tactics in Cognitive Science.* New York: Erlbaum, in press.

Vere, S.A. Induction of concepts in the predicate calculus. *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, Tbilisi, USSR, 1975, 281-287.

Vere, S.A. Inductive learning of relational productions. In D.A. Waterman & F. Hayes-Roth (eds.), *Pattern directed inference systems.* New York: Academic Press, 1978.

Waldinger, R. Achieving several goals simultaneously (Tech. No. 107). Menlo Park, CA: Stanford Research Institute, Artificial Intelligence group, 1975.

Waters, R.C. Automatic analysis of the logical structure of programs. (Tech. Rep. 492), Cambridge, MA: Massachusetts Institute of Technology, Artificial Intelligence Lab., 1978.

Welchons, A.M., Krickenberger, W.R., Paerson, H.R., Duffy, A.G. & McCaffery, J.M. *Algebra, Book I.* Lexington, MA: Ginn & Co., 1981.

Welford, A.T. *Skilled Performance: Perceptual and motor skills.* Glenview, IL: Scott, Foresman and Co., 1976.

Williams, M.D., Hollan, J.D. & Stevens, A.L. Human reasoning about a simple physical system (CIS-7). Palo Alto, CA: Xerox Palo Alto Research Center, 1981.

Winston, P.H. Learning structural descriptions from examples. In P.H. Winston (ed.), *The Psychology of Computer Vision*. New York: McGraw-Hill, 1975.

Winston, P.H. *Artificial Intelligence*, Reading, MA: Addison-Wesley, 1977.

Winston, P.H. Learning by understanding analogies (Rep. No. AIM-520). Cambridge, MA: Massachusetts Institute of Technology, Artificial Intelligence laboratory, 1979.

Woods, W.A. Transition network grammars for natural language analysis. *Communications of the ACM*, 1970, *13*, 10, 591-606.

Yasuhara, A. *Recursive Function Theory and Logic*. New York: Academic, 1971.

Young, R.M. & O'Shea, T. Errors in children's subtraction. *Cognitive Science*, 1981, *5*, 153–177.

Appendix 1
A Bug Glossary

0-N=0/AFTER/BORROW
When a column has a 1 that was changed to a 0 by a previous borrow, the student writes 0 as the answer to that column. (914 - 486 = 508)

0-N=0/EXCEPT/AFTER/BORROW
Thinks 0-N is 0 except when the column has been borrowed from. (906 - 484 = 502)

0-N=N/AFTER/BORROW
When a column has a 1 that was changed to a 0 by a previous borrow, the student writes the bottom digit as the answer to that column. (512 - 136 = 436)

0-N=N/EXCEPT/AFTER/BORROW
Thinks 0-N is N except when the column has been borrowed from. (906 - 484 = 582)

1-1=0/AFTER/BORROW
If a column starts with 1 in both top and bottom and is borrowed from, the student writes 0 as the answer to that column. (812 - 518 = 304)

1-1=1/AFTER/BORROW
If a column starts with 1 in both top and bottom and is borrowed from, the student writes 1 as the answer to that column. (812 - 518 = 314)

ADD/BORROW/CARRY/SUB
The student adds instead of subtracts but he subtracts the carried digit instead of adding it. (54 - 38 = 72)

ADD/BORROW/DECREMENT
Instead of decrementing the student adds 1, carrying to the next column if necessary.

$$\begin{array}{rrrr} & 8 & 6 & 3 \\ - & 1 & 3 & 4 \\ \hline & 7 & 4 & 9 \end{array} \qquad \begin{array}{rrrr} & 8 & 9 & 3 \\ - & 1 & 0 & 4 \\ \hline & 8 & 0 & 9 \end{array}$$

ADD/BORROW/DECREMENT/WITHOUT/CARRY
Instead of decrementing the student adds 1. If this addition results in 10 the student does not carry but simply writes both digits in the same space.

$$\begin{array}{rrrr} & 8 & 6 & 3 \\ - & 1 & 3 & 4 \\ \hline & 7 & 4 & 9 \end{array} \qquad \begin{array}{rrrr} & 8 & 9 & 3 \\ - & 1 & 0 & 4 \\ \hline & 7 & 10 & 9 \end{array}$$

ADD/INSTEADOF/SUB
The student adds instead of subtracts. (32 - 15 = 47)

ADD/LR/DECREMENT/ANSWER/CARRY/TO/RIGHT Adds columns from left to right instead of subtracts. Before writing the column's answer, it is decremented and truncated to the units digit. A one is added into the next column to the right. (411 - 215 = 527)

ADD/NOCARRY/INSTEADOF/SUB
The student adds instead of subtracting. If carrying is required, he does not add the carried digit. (47 - 25 = 62)

ALWAYS/BORROW
The student borrows in every column regardless of whether it is necessary. (488 - 229 = 1159)

ALWAYS/BORROW/LEFT
The student borrows from the leftmost digit instead of borrowing from the digit immediately to the left. (733 - 216 = 427)

BLANK/INSTEADOF/BORROW
When a borrow is needed the student simply the skips the column and goes on to the next. (425 - 283 = 2 2)

BORROW/ACROSS/TOP/SMALLER/DECREMENTING/TO

When decrementing a column in which the top is smaller than the bottom, the student adds 10 to the top digit, decrements the column being borrowed into and borrows from the next column to the left. Also the student skips any column which has a 0 over a 0 or a blank in the borrowing process.

$$\begin{array}{r} 183 \\ -\ 95 \\ \hline 97 \end{array} \qquad \begin{array}{r} 513 \\ -\ 268 \\ \hline 254 \end{array}$$

BORROW/ACROSS/ZERO

When borrowing across a 0, the student skips over the 0 to borrow from the next column. If this causes him to have to borrow twice he decrements the same number both times.

$$\begin{array}{r} 904 \\ -\ 7 \\ \hline 807 \end{array} \qquad \begin{array}{r} 904 \\ -\ 237 \\ \hline 577 \end{array}$$

BORROW/ACROSS/ZERO/OVER/BLANK

When borrowing across a 0 over a blank, the student skips to the next column to decrement. (402 - 6 = 306)

BORROW/ACROSS/ZERO/OVER/ZERO

Instead of borrowing across a 0 that is over a 0, the student does not change the 0 but decrements the next column to the left instead. (802 - 304 = 308)

BORROW/ADD/DECREMENT/INSTEADOF/ZERO

Instead of borrowing across a 0, the student changes the 0 to 1 and doesn't decrement any column to the left. (307 - 108 = 219)

BORROW/ADD/IS/TEN

The student changes the number that causes the borrow into 10 instead of adding 10 to it. (83 - 29 = 51)

BORROW/DECREMENTING/TO/BY/EXTRAS

When there is a borrow across 0's, the student does not add 10 to the column he is doing but instead adds 10 minus the number of 0's borrowed across.

$$\begin{array}{r} 308 \\ -\ 139 \\ \hline 168 \end{array} \qquad \begin{array}{r} 3008 \\ -\ 1359 \\ \hline 1647 \end{array}$$

BORROW/DIFF/0-N=N&SMALL-LARGE=0

The student doesn't borrow. For columns of the form 0 - N, he writes N as the answer. Otherwise he writes 0. (304 - 179 = 270)

BORROW/DON'T/DECREMENT/TOP/SMALLER

The student will not decrement a column if the top number is smaller than the bottom number.

$$\begin{array}{r} 732 \\ -\ 484 \\ \hline 258 \\ \text{Wrong} \end{array} \qquad \begin{array}{r} 732 \\ -\ 434 \\ \hline 298 \\ \text{Correct} \end{array}$$

BORROW/DON'T/DECREMENT/UNLESS/BOTTOM/SMALLER

The student will not decrement a column unless the bottom number is smaller than the top number.

$$\begin{array}{r} 732 \\ -\ 484 \\ \hline 258 \end{array} \qquad \begin{array}{r} 732 \\ -\ 434 \\ \hline 308 \end{array}$$

BORROW/FROM/ALL/ZERO

Instead of borrowing across 0's, the student changes all the 0's to 9's but does not continue borrowing from the column to the left. (3006 - 1807 = 2199)

BORROW/FROM/BOTTOM

The student borrows from the bottom row instead of the top one.

$$\begin{array}{r} 87 \\ -\ 28 \\ \hline 79 \end{array} \qquad \begin{array}{r} 827 \\ -\ 208 \\ \hline 839 \end{array}$$

BORROW/FROM/BOTTOM/INSTEADOF/ZERO

When borrowing from a column of the form 0 - N, the student decrements the bottom number instead of the 0.

$$\begin{array}{r} 608 \\ -\ 249 \\ \hline 379 \end{array} \qquad \begin{array}{r} 108 \\ -\ 49 \\ \hline 79 \end{array}$$

BORROW/FROM/LARGER

When borrowing, the student decrements the larger digit in the column regardless of whether it is on the top or the bottom. (872 - 294 = 598)

BORROW/FROM/ONE/IS/NINE

When borrowing from a 1, the student treats the 1 as if it were 10, decrementing it to a 9. (316 - 139 = 267)

BORROW/FROM/ONE/IS/TEN

When borrowing from a 1, the student changes the 1 to 10 instead of to 0. (414 - 277 = 237)

BORROW/FROM/ZERO

Instead of borrowing across a 0, the student changes the 0 to 9 but does not continue borrowing from the column to the left.

$$\begin{array}{r} 306 \\ -\,187 \\ \hline 219 \end{array} \qquad \begin{array}{r} 3006 \\ -\,1807 \\ \hline 1299 \end{array} \qquad \begin{array}{r} 103 \\ -\;\;45 \\ \hline 158 \end{array}$$

BORROW/FROM/ZERO&LEFT/OK

Instead of borrowing across a 0, the student changes the 0 to 9 but does not continue borrowing from the column ot the left. However if the digit to the left of the 0 is over a blank then the student does the correct thing.

$$\begin{array}{r} 306 \\ -\,187 \\ \hline 219 \\ \text{Wrong} \end{array} \qquad \begin{array}{r} 3006 \\ -\,1807 \\ \hline 1299 \\ \text{Wrong} \end{array} \qquad \begin{array}{r} 103 \\ -\;\;45 \\ \hline 58 \\ \text{Correct} \end{array} \qquad \begin{array}{r} 203 \\ -\;\;45 \\ \hline 158 \\ \text{Correct} \end{array}$$

BORROW/FROM/ZERO/IS/TEN

When borrowing across 0, the student changes the 0 to 10 and does not decrement any digit to the left. (604 - 235 = 479)

BORROW/IGNORE/ZERO/OVER/BLANK

When borrowing across a 0 over a blank, the student treats the column with the zero as if it weren't there.

$$\begin{array}{r} 505 \\ -\;\;\;\;7 \\ \hline 4\;\;\,8 \\ \text{Wrong} \end{array} \qquad \begin{array}{r} 508 \\ -\;\;\;\;7 \\ \hline 501 \\ \text{Correct} \end{array}$$

BORROW/INTO/ONE=TEN

When a borrow is caused by a 1, the student changes the 1 to a 10 instead of adding 10 to it. (71 - 38 = 32)

BORROW/NO/DECREMENT

When borrowing the student adds 10 correctly but doesn't change any column to the left. (62 - 44 = 28)

BORROW/NO/DECREMENT/EXCEPT/LAST

Decrements only in the last column of the problem. (6262 - 4444 = 1828)

BORROW/ONCE/THEN/SMALLER/FROM/LARGER

The student will borrow only once per exercise. From then on he subtracts the smaller from the larger digit in each column regardless of their positions. (7127 - 2389 = 5273)

BORROW/ONCE/WITHOUT/RECURSE

The student will borrow only once per problem. After that, if another borrow is required the student adds the 10 correctly but does not decrement. If there is a borrow across a 0, the student changes the 0 to 9 but does not decrement the digit to the left of the 0.

$$\begin{array}{r} 535 \\ -\,278 \\ \hline 357 \end{array} \qquad \begin{array}{r} 408 \\ -\,239 \\ \hline 269 \end{array}$$

BORROW/ONLY/FROM/TOP/SMALLER

When borrowing, the student tries to find a column in which the top number is smaller than the bottom. If there is one, he decrements that, otherwise he borrows correctly. (9283 - 3566 = 5627)

BORROW/ONLY/ONCE

When there are several adjacent borrows, the student decrements only with the first borrow. (535 - 278 = 357)

BORROW/SKIP/EQUAL

When decrementing, the student skips over columns in which the top digit and the bottom digit are the same. (923 - 427 = 406)

BORROW/TEN/PLUS/NEXT/DIGIT/INTO/ZERO
When a borrow is caused by a 0, the student does not add 10 correctly. What he does instead is add 10 plus the digit in the next column to the left. (50 - 38 = 17)

BORROW/TREAT/ONE/AS/ZERO
When borrowing from 1, the student treats the 1 as if it were 0; that is, he changes the 1 to 9 and decrements the number to the left of the 1. (313 - 159 = 144)

BORROW/UNIT/DIFF
The student borrows the difference between the top digit and the bottom digit of the current column. In other words, he borrows just enough to do the subtraction, which then always results in 0. (86 - 29 = 30)

BORROW/WONT/RECURSE
Instead of borrowing across a 0, the student stops doing the exercise. (8035 - 2662 = 3)

BORROWED/FROM/DONT/BORROW
When there are two borrows in a row the student does the first borrow correctly but with the second borrow he does not decrement (he does add 10 correctly). (143 - 88 = 155)

CAN'T/SUBTRACT
The student skips the entire problem. (8 - 3 =)

COPY/TOP/IN/LAST/COLUMN/IF/BORROWED/FROM
After borrowing from the last column, the student copies top digit as the answer (80 - 34 = 76).

DECREMENT/ALL/ON/MULTIPLE/ZERO
When borrowing across a 0 and the borrow is caused by 0, the student changes the right 0 to 9 instead of 10. (600 - 142 =457)

DECREMENT/BY/ONE/PLUS/ZEROS
When there is a borrow across zero, decrements the number to the left of the zero(s) by an extra one for every zero borrowed across. (4005 - 6 = 1999)

DECREMENT/BY/TWO/OVER/TWO
When borrowing from a column of the form N - 2, the student decrements the N by 2 instead of 1. (83 - 29 =44)

DECREMENT/LEFTMOST/ZERO/ONLY
When borrowing across two or more 0's, the student changes the leftmost of the row of 0's to 9 but changes the other 0's to 10's. He will give answers like: (1003 - 958 = 1055)

DECREMENT/MULTIPLE/ZEROS/BY/NUMBER/TO/LEFT
When borrowing across 0's, the student changes the leftmost 0 to a 9, changes the next 0 to 8, etc. (8002 - 1714 = 6278)

DECREMENT/MULTIPLE/ZEROS/BY/NUMBER/TO/RIGHT
When borrowing across 0's, the student changes the rightmost 0 to a 9, changes the next 0 to 8, etc. (8002 - 1714 = 6188)

DECREMENT/ON/FIRST/BORROW
The first column that requires a borrow is decremented before the column subtract is done. (832 - 265 = 566)

DECREMENT/ONE/TO/ELEVEN
Instead of decrementing a 1, the student changes the 1 to an 11. (314 - 6 = 2118)

DECREMENT/TOP/LEQ/IS/EIGHT
When borrowing from 0 or 1, changes the 0 or 1 to 8; does not decrement digit to the left of the 0 or 1. (4013 - 995 = 3778)

DIFF/0-N=0
When the student encounters a column of the form 0 - N, he doesn't borrow; instead he writes 0 as the column answer. (40 - 21 = 20)

DIFF/0-N=N
When the student encounters a column of the form 0 - N, he doesn't borrow. Instead he writes N as the answer. (80 - 27 = 67)

DIFF/0-N=N/WHEN/BORROW/FROM/ZERO
When borrowing across a 0 and the borrow is caused by a 0, the student doesn't borrow. Instead he writes the bottom number as the column answer. He will borrow correctly in the next column or in other circumstances.

$$\begin{array}{rccc} & 1 & 0 & 0 \\ - & & 3 & 2 \\ \hline & & 7 & 2 \end{array} \qquad \begin{array}{rccc} & 4 & 0 & 0 \\ - & 2 & 4 & 8 \\ \hline & 1 & 6 & 8 \end{array}$$

DIFF/1-N=1
When a column has the form 1 - N, the student writes 1 as the column answer. (51 - 27 = 31)

DIFF/N-0=0
The student thinks that N - 0 is 0. (57 - 20 = 30)

DIFF/N-N=N
Whenever there is a column that has the same number on the top and the bottom, the student writes that number as the answer. (83 - 13 = 73)

DOESNT/BORROW
The student stops doing the exercise when a borrow is required. (833 - 262 = 1)

DONT/DECREMENT/SECOND/ZERO
When borrowing across a 0 and the borrow is caused by a 0, the student changes the 0 he is borrowing across into a 10 instead of a 9. (700 - 258 =452)

DONT/DECREMENT/ZERO
When borrowing across a 0, the student changes the 0 to 10 instead of 9. (506 - 318 = 198)

DONT/DECREMENT/ZERO/OVER/BLANK
The student will not borrow across a zero that is over a blank. (305 - 9 = 306)

DONT/DECREMENT/ZERO/OVER/ZERO
The student will not borrow across a zero that is over a zero. (305 - 107 = 308)

DONT/DECREMENT/ZERO/UNTIL/BOTTOM/BLANK
When borrowing across a 0, the student changes the 0 to a 10 instead of a 9 unless the 0 is over a blank, in which case he does the correct thing.

$$\begin{array}{rrr} & 5\ 0\ 6 \\ - & 3\ 1\ 8 \\ \hline & 1\ 9\ 8 \end{array} \qquad \begin{array}{rrr} & 3\ 0\ 4 \\ - & 9 \\ \hline & 2\ 9\ 5 \end{array}$$

Wrong Correct

DONT/WRITE/ZERO
Doesn't write zeros in the answer. (24 - 14 = 1)

DOUBLE/DECREMENT/ONE
When borrowing from a 1, the student treats the 1 as a 0 (changes the 1 to 9 and continues borrowing to the left. (813 - 515 = 288)

FORGET/BORROW/OVER/BLANKS
The student doesn't decrement a number that is over a blank. (347 - 9 = 348)

IGNORE/LEFTMOST/ONE/OVER/BLANK
When the left column of the exercise has a 1 that is over a blank, the student ignores that column. (143 - 22 = 21)

IGNORE/ZERO/OVER/BLANK
Whenever there is column that has a 0 over a blank, the student ignores that column. (907 - 5 = 9 2)

INCREMENT/OVER/LARGER
When borrowing from a column in which the top is smaller than the bottom, the student increments instead of decrementing. (833 -277 = 576)

INCREMENT/ZERO/OVER/BLANK
When borrowing across a 0 over a blank, the student increments the 0 instead of decrementing. (402 - 5 = 416)

N-9=N-1/AFTER/BORROW
If a column is of the form N - 9 and has been borrowed from, when the student does that column he subtracts 1 instead of subtracting 9. (834 - 796 = 127)

N-N/AFTER/BORROW/CAUSES/BORROW
Borrows with columns of the form N-N if the column has been borrowed from. (953 - 147 = 7106)

N-N/CAUSES/BORROW
Borrows with columns of the form N-N. (953 - 152 = 7101)

N-N=1/AFTER/BORROW
If a column had the form N - N and was borrowed from, the student writes 1 as the answer to that column. (944 - 348 = 616)

N-N=9/PLUS/DECREMENT
When a column has the same number on the top and the bottom, the student writes 9 as the answer and decrements the next column to the left even though borrowing is not necessary.
(94 - 34 = 59)

ONCE/BORROW/ALWAYS/BORROW
Once a student has borrowed, he continues to borrow in every remaining column in the exercise. (488 - 229 = 1159)

QUIT/WHEN/BOTTOM/BLANK
When the bottom number has fewer digits than the top number, the student quits as soon as the bottom number runs out.
(439 - 4 = 5)

SIMPLE/PROBLEM/STUTTER/SUBTRACT
When the bottom number is a single digit and the top number has two or more digits, the student repeatedly subtracts the single bottom digit from each digit in the top number.
(348 - 2 = 126)

SMALLER/FROM/LARGER
The student doesn't borrow; in each column he subtracts the smaller digit from the larger one.
(81 - 38 = 57)

SMALLER/FROM/LARGER/INSTEAD/OF/BORROW/FROM/ZERO
The student does not borrow across 0. Instead he will subtract the smaller from the larger digit.

$$\begin{array}{rrr} 3 & 0 & 6 \\ - & & 8 \\ \hline 3 & 0 & 2 \end{array} \qquad \begin{array}{rrrr} & 3 & 0 & 6 \\ - & 1 & 4 & 8 \\ \hline & 1 & 6 & 2 \end{array}$$

SMALLER/FROM/LARGER/WHEN/BORROWED/FROM
When there are two borrows in a row, the student does the first one correctly but for the second one he does not borrow; instead he subtracts the smaller from the larger digit, regardless of order. (824 - 157 = 747)

SMALLER/FROM/LARGER/WITH/BORROW
When borrowing the student decrements correctly, then subtracts the smaller digit from the larger as if he had not borrowed at all. (73 - 24 = 411)

STOPS/BORROW/AT/MULTIPLE/ZERO
Instead of borrowing across several 0's, the student adds 10 to the column he's doing but doesn't change any column to the left. (4004 - 9 = 4005)

STOPS/BORROW/AT/SECOND/ZERO
When borrowing across several 0's, changes the right 0 to 9 but not the other 0's. (4004 - 9 = 4095)

STOPS/BORROW/AT/ZERO
Instead of borrowing across a 0, the student adds 10 to the column he's doing but doesn't decrement from a column to the left. (404 - 187 = 227)

STUTTER/SUBTRACT
When there are blanks in the bottom number, the student subtracts the leftmost digit of the bottom number in every column that has a blank. (4369 - 22 = 2147)

SUB/BOTTOM/FROM/TOP
The student always subtracts the top digit from the bottom digit. If the bottom digit is smaller, he decrements the top digit and adds 10 to the bottom before subtracting. If the bottom digit is zero, however, he writes the top digit in the answer. If the top digit is 1 greater than the bottom he writes 9. He will give answers like this. (4723 - 3065 = 9742)

SUB/COPY/LEAST/BOTTOM/MOST/TOP
The student does not subtract. Instead he copies digits from the exercise to fill in the answer space. He copies the leftmost digit from the top number and the other digits from the bottom number. He will give answers like this: (648 - 231 = 631)

SUB/ONE/OVER/BLANKS
When there are blanks in the bottom number, the student subtracts 1 from the top digit.
(548 - 2 = 436)

TREAT/TOP/ZERO/AS/NINE
In a 0-N column, the student doesn't borrow. Instead he treats the 0 as if it were a 9. (30 - 4 = 39)

TREAT/TOP/ZERO/AS/TEN
In a 0-N column, the student adds 10 to it correctly but doesn't change any column to the left. (40 - 27 = 23)

X-N=0/AFTER/BORROW
If a column has been borrowed from, the student writes zero as its answer. (234 - 115 = 109)

X-N=N/AFTER/BORROW
If a column has been borrowed from, the student writes the bottom digit as its answer. (234 - 165 = 169)

ZERO/AFTER/BORROW
When a column requires a borrow, the student decrements correctly but writes 0 as the answer.
(65 - 48 = 10)

ZERO/INSTEAD/OF/BORROW/FROM/ZERO
The student won't borrow if he has to borrow across 0. Instead he will write 0 as the answer to the column requiring the borrow.

$$\begin{array}{r} 7\ 0\ 2 \\ -\quad\ \ 8 \\ \hline 7\ 0\ 0 \end{array} \qquad \begin{array}{r} 7\ 0\ 2 \\ -\ 3\ 4\ 8 \\ \hline 3\ 6\ 0 \end{array}$$

ZERO/INSTEADOF/BORROW
The student doesn't borrow; he writes 0 as the answer instead. (42 -16 = 30)

Appendix 2
Observed Bug Sets

The diagnoses of all tests of all students analyzed by Debuggy fall into the following categories:

No errors	112	(10%)
Errors due to slips alone	239	(20%)
Errors due to bugs (and slips)	375	(33%)
Unanalyzable	421	(37%)
total	1147	(100%)

The diagnoses of the students that were analyzed as having bugs are shown, ordered by their frequency of occurrence. Diagnoses consisting of more than one bug are shown in parentheses. There are 134 distinct diagnoses, of which only 35 occurred more than once. However, these 35 diagnoses account for 276 of the 375 cases (74%).

The diagnoses in the appendices sometimes contain *coercions*. A coercion is a modifier that is included in a diagnosis to improve the fit of the bugs to the student's errors. Most often, these slightly perturb the definitions of bugs. For example, certain bugs modify the procedure so that on occasion it will write column answers that are greater than 9. Some students who have these bugs apparently know from addition that there should only be one answer digit per column, so they only write the units digit. To capture this, the coercion !Write-Units-Digit-Only is added to the diagnoses of such students by Debuggy. Coercions can easily be picked out because their names have exclamation points as prefixes. For more on coercions, see (Burton, 1981).

103 occurrences
(Smaller-From-Larger)

34 occurrence
(Stops-Borrow-At-Zero)

13 occurrences
(Borrow-Across-Zero)

10 occurrences
(Borrow-From-Zero)
(Borrow-No-Decrement)

7 occurrences
(Stops-Borrow-At-Zero Diff-0-N=N)

6 occurrences
(Always-Borrow-Left)
(Borrow-Across-Zero !Touched-0-Is-Ten)
(Borrow-Across-Zero Diff-0-N=N)
(Borrow-Across-Zero-Over-Zero Borrow-Across-Zero-Over-Blank)

(Stops-Borrow-At-Zero Borrow-Once-Then-Smaller-From-Larger Diff-0-N=N)

5 occurrences
(Borrow-No-Decrement Diff-0-N=N)

4 occurrences
(0-N=N-Except-After-Borrow)
(Borrow-Across-Zero Diff-0-N=0)
(Diff-0-N=N Zero-Insteadof-Borrow)
(Borrow-No-Decrement-Except-Last)
(Don't-Decrement-Zero-Over-Blank)
(Quit-When-Bottom-Blank Smaller-From-Larger)

3 occurrences
(Borrow-Into-One=Ten Stops-Borrow-At-Zero)
(Decrement-All-On-Multiple-Zero)
(Decrement-Multiple-Zeros-By-Number-To-Right)
(Don't-Decrement-Zero)
(Smaller-From-Larger Ignore-Leftmost-One-Over-Blank)

2 occurrences
(0-N=0-After-Borrow)
(Borrow-Across-Second-Zero)
(Borrow-Across-Top-Smaller-Decrementing-To)
(Borrow-Don't-Decrement-Top-Smaller)
(Borrow-Don't-Decrement-Unless-Bottom-Smaller)
(Borrow-Only-From-Top-Smaller Borrow-Across-Zero-Over-Zero Borrow-Across-Zero-Over-Blank)
(Smaller-From-Larger-Instead-Of-Borrow-From-Zero Borrow-Once-Then-Smaller-From-Larger
Diff-0-N=N)
(Smaller-From-Larger-Insteadof-Borrow-Unless-Bottom-Smaller)
(Stops-Borrow-At-Multiple-Zero)
(Stops-Borrow-At-Zero Smaller-From-Larger-When-Borrowed-From)
(Stops-Borrow-At-Zero Diff-0-N=N Smaller-From-Larger-When-Borrowed-From)
(Stutter-Subtract)

1 occurrence
(!Only-Write-Units-Digit N-N-After-Borrow-Causes-Borrow)
(!Only-Write-Units-Digit Stops-Borrow-At-Multiple-Zero N-N-After-Borrow-Causes-Borrow)
(!Sub-Units-Special Borrow-Across-Zero Smaller-From-Larger)
(!Write-Left-Ten Smaller-From-Larger Diff-0-N=0)
(!Write-Left-Ten Borrow-Across-Second-Zero Diff-0-N=N)
(!Write-Left-Ten Forget-Borrow-Over-Blanks Diff-0-N=N)
(!Touched-0-N=N Borrow-Across-Zero Diff-N-0=0)
(!Touched-0-N=N Borrow-Across-Zero Borrow-Once-Then-Smaller-From-Larger)
(!Touched-0-N=N Borrow-Across-Zero Borrow-Across-Second-Zero
Smaller-From-Larger-When-Borrowed-From)
(0-N=N-After-Borrow Borrow-Across-Zero-Over-Zero Borrow-Across-Zero-Over-Blank)
(0-N=N-After-Borrow N-N=1-After-Borrow Smaller-From-Larger-Instead-Of-Borrow-From-Zero)
(0-N=N-After-Borrow)
(0-N=N-Except-After-Borrow 1-1=0-After-Borrow)

(0-N=N-Except-After-Borrow 1-1=1-After-Borrow)
(1-1=0-After-Borrow)
(Add-Insteadof-Sub)
(Add-Lr-Decrement-Answer-Carry-To-Right)
(Blank-Insteadof-Borrow Diff-0-N=N)
(Borrow-Across-Second-Zero Don't-Write-Zero)
(Borrow-Across-Second-Zero Borrow-Skip-Equal)
(Borrow-Across-Zero 0-N=0-Except-After-Borrow)
(Borrow-Across-Zero 1-1=0-After-Borrow)
(Borrow-Across-Zero Borrow-Once-Then-Smaller-From-Larger 0-N=N-Except-After-Borrow)
(Borrow-Across-Zero Sub-One-Over-Blank 0-N=N-After-Borrow 0-N=N-Except-After-Borrow)
(Borrow-Across-Zero Borrow-Skip-Equal)
(Borrow-Across-Zero Quit-When-Bottom-Blank 0-N=0-After-Borrow)
(Borrow-Across-Zero Forget-Borrow-Over-Blanks Diff-0-N=N)
(Borrow-Across-Zero Ignore-Leftmost-One-Over-Blank Borrow-Skip-Equal)
(Borrow-Across-Zero !Touched-0-N=N)
(Borrow-Across-Zero-Over-Zero 0-N=N-Except-After-Borrow 1-1=0-After-Borrow)
(Borrow-Across-Zero-Over-Zero)
(Borrow-Don't-Decrement-Unless-Bottom-Smaller X-N=0-After-Borrow)
(Borrow-Don't-Decrement-Unless-Bottom-Smaller Don't-Write-Zero)
(Borrow-From-All-Zero)
(Borrow-From-Bottom-Insteadof-Zero Diff-0-N=N)
(Borrow-From-One-Is-Nine Borrow-From-Zero Diff-0-N=N-When-Borrow-From-Zero)
(Borrow-From-One-Is-Nine Borrow-From-Zero Don't-Decrement-Zero-Over-Blank)
(Borrow-From-One-Is-Ten Borrow-From-Zero-Is-Ten Borrow-Only-Once)
(Borrow-From-Zero 0-N=0-After-Borrow)
(Borrow-From-Zero 0-N=N-After-Borrow)
(Borrow-From-Zero&Left-Ten-Ok 0-N=N-After-Borrow)
(Borrow-From-Zero&Left-Ten-Ok)
(Borrow-From-Zero-Is-Ten)
(Borrow-Into-One=Ten Decrement-Multiple-Zeros-By-Number-To-Left)
(Borrow-Into-One=Ten Decrement-Multiple-Zeros-By-Number-To-Right
Borrow-Across-Zero-Over-Zero)
(Borrow-No-Decrement Diff-0-N=0)
(Borrow-No-Decrement Smaller-From-Larger-Except-Last
Smaller-From-Larger-Insteadof-Borrow-Unless-Bottom-Smaller Diff-0-N=N)
(Borrow-No-Decrement Sub-One-Over-Blank)
(Borrow-No-Decrement-Except-Last Treat-Top-Zero-As-Ten)
(Borrow-No-Decrement-Except-Last Decrement-Top-Leq-Is-Eight X-N=N-After-Borrow)
(Borrow-Only-From-Top-Smaller)
(Borrow-Only-From-Top-Smaller 0-N=N-After-Borrow)
(Borrow-Treat-One-As-Zero N-N=1-After-Borrow Don't-Decrement-Zero-Over-Blank)
(Borrow-Unit-Diff Only-Do-Units)
(Can't-Subtract)
(Decrement-All-On-Multiple-Zero Double-Decrement-One)
(Decrement-Leftmost-Zero-Only)
(Decrement-Multiple-Zeros-By-Number-To-Left)
(Decrement-Top-Leq-Is-Eight)
(Diff-0-N=0 Diff-N-0=0 Stops-Borrow-At-Zero)
(Diff-0-N=0 Diff-N-0=0 Doesn't-Borrow-Except-Last

Smaller-From-Larger-Insteadof-Borrow-Unless-Bottom-Smaller)
(Diff-0-N=0 Diff-N-0=0 Smaller-From-Larger-Except-Last Smaller-From-Larger-Insteadof-Borrow-Unless-Bottom-Smaller)
(Diff-0-N=N)
(Diff-0-N=N Diff-N-0=0)
(Diff-0-N=N-When-Borrow-From-Zero Don't-Decrement-Zero)
(Diff-N-0=0 Smaller-From-Larger Diff-0-N=0)
(Don't-Decrement-Zero Borrow-Across-Second-Zero)
(Don't-Decrement-Zero 1-1=0-After-Borrow)
(Don't-Decrement-Zero Decrement-One-To-Eleven)
(Don't-Decrement-Zero-Until-Bottom-Blank Borrow-Across-Zero-Over-Zero)
(Don't-Write-Zero)
(Double-Decrement-One)
(Double-Decrement-One Smaller-From-Larger-When-Borrowed-From)
(Forget-Borrow-Over-Blanks)
(Forget-Borrow-Over-Blanks Borrow-Don't-Decrement-Top-Smaller Borrow-Skip-Equal)
(Ignore-Leftmost-One-Over-Blank Decrement-All-On-Multiple-Zero)
(N-N-Causes-Borrow)
(N-N=1-After-Borrow 0-N=N-Except-After-Borrow)
(N-N=1-After-Borrow)
(Simple-Problem-Stutter-Subtract)
(Smaller-From-Larger Diff-N-N=N Diff-0-N=0)
(Smaller-From-Larger Diff-0-N=0)
(Smaller-From-Larger-Except-Last Decrement-All-On-Multiple-Zero)
(Smaller-From-Larger-Instead-Of-Borrow-From-Zero Diff-0-N=N Smaller-From-Larger-When-Borrowed-From)
(Smaller-From-Larger-Instead-Of-Borrow-From-Zero Borrow-Once-Then-Smaller-From-Larger)
(Smaller-From-Larger-Insteadof-Borrow-Unless-Bottom-Smaller 0-N=N-Except-After-Borrow)
(Smaller-From-Larger-Insteadof-Borrow-Unless-Bottom-Smaller Top-Instead-Of-Borrow-From-Zero Diff-0-N=N)
(Stops-Borrow-At-Zero 0-N=0-Except-After-Borrow 1-1=0-After-Borrow)
(Stops-Borrow-At-Zero Borrow-Across-Zero-Over-Zero 1-1=0-After-Borrow)
(Stops-Borrow-At-Zero 1-1=1-After-Borrow)
(Stops-Borrow-At-Zero Diff-0-N=0)
(Stops-Borrow-At-Zero Ignore-Leftmost-One-Over-Blank)
(Stops-Borrow-At-Zero 0-N=0-After-Borrow)
(Stops-Borrow-At-Zero Borrow-Once-Then-Smaller-From-Larger)
(Stops-Borrow-At-Zero 1-1=0-After-Borrow)
(Stops-Borrow-At-Zero Diff-0-N=N Don't-Write-Zero)
(Sub-Bottom-From-Top)
(Sub-Copy-Least-Bottom-Most-Top)
(Zero-Insteadof-Borrow)

Appendix 3
Bug Occurrence Frequencies

This appendix lists each bug and coercion in the Debuggy's database (see the previous appendix for an explanation of coercions). It indicates how many times the bug has occurred, if any, in Debuggy's analyses of the Southbay data. The first column, labelled "alone" is the number of times the bug occurred alone, as the only element of the diagnosis. The second column, labelled "cmd." is the number of times the bug occurred as part of a multi-bug diagnosis, or "compound" bug as it was called in (Brown & Burton, 1978). The third column, labelled "gen.", has a "$" mark if the bug was generated by Sierra during the Southbay run. Thus, for example, the bug 1-1=0-After-Borrow occurred once alone, and seven times as part of a larger diagnosis, but is not one of the bugs that Sierra predicted. Rows that would be all zeros have been left blank to highlight those bugs in the data base which never occurred in these studies. The data come from the reanalysis that was performed after the new bugs generated by Sierra were entered in the database. There are 128 bugs and 15 coercions in the database. Of these, 75 bugs and 5 coercions occurred at least once. Sierra generated 49 bugs and 6 coercions.

alone	cmd.	gen.	Coercion
			!Borrow-Diff-Abs-Over-Blank
			!Forget-To-Write-Units-Digit
			!Last-Column-Special-Sub
			!Last-Full-Column-Special-Sub
			!N-0=N-Always
0	2		!Only-Write-Units-Digit
0	1		!Sub-Units-Special
0	3		!Write-Left-Ten
			!Zero-Minus-Blank-Is-Zero
		$	!Touched-0-N=0
		$	!Touched-0-N=Blank
0	4	$	!Touched-0-N=N
0	6	$	!Touched-0-Is-Ten
		$	!Touched-0-Is-Quit
		$	!Touched-Double-Zero-Is-Quit

alone	cmd.	gen.	Bug
2	3		0-N=0-After-Borrow
0	2		0-N=0-Except-After-Borrow
1	6		0-N=N-After-Borrow
4	7		0-N=N-Except-After-Borrow

1	7		1-1=0-After-Borrow
0	2		1-1=1-After-Borrow
			Add-Borrow-Decrement
			Add-Borrow-Decrement-Without-Carry
1	0		Add-Insteadof-Sub
1	0		Add-Lr-Decrement-Answer-Carry-To-Right
			Add-Nocarry-Insteadof-Sub
			Always-Borrow
6	0	$	Always-Borrow-Left
		$	Blank-Instead-Of-Borrow-From-Zero
0	1	$	Blank-Insteadof-Borrow
		$	Blank-Insteadof-Borrow-Except-Last
		$	*Blank-Insteadof-Borrow-From-Double-Zero
		$	Blank-Insteadof-Borrow-Unless-Bottom-Smaller
		$	*Blank-With-Borrow
2	6	$	Borrow-Across-Second-Zero
2	0		Borrow-Across-Top-Smaller-Decrementing-To
13	29	$	Borrow-Across-Zero
0	9		Borrow-Across-Zero-Over-Blank
1	13		Borrow-Across-Zero-Over-Zero
		$	Borrow-Add-Decrement-Insteadof-Zero
			Borrow-Add-Is-Ten
			Borrow-Decrementing-To-By-Extras
2	1		Borrow-Don't-Decrement-Top-Smaller
2	2	$	Borrow-Don't-Decrement-Unless-Bottom-Smaller
1	0		Borrow-From-All-Zero
			Borrow-From-Bottom
0	1		Borrow-From-Bottom-Insteadof-Zero
			Borrow-From-Larger
0	2	$	Borrow-From-One-Is-Nine
0	1	$	Borrow-From-One-Is-Ten
10	4	$	Borrow-From-Zero
1	1		Borrow-From-Zero&Left-Ten-Ok
1	1	$	Borrow-From-Zero-Is-Ten
			Borrow-Ignore-Zero-Over-Blank
0	6		Borrow-Into-One=Ten
10	8	$	Borrow-No-Decrement
4	2	$	Borrow-No-Decrement-Except-Last
0	12		Borrow-Once-Then-Smaller-From-Larger
			Borrow-Once-Without-Recurse
1	3		Borrow-Only-From-Top-Smaller
0	1		Borrow-Only-Once
0	4		Borrow-Skip-Equal
			Borrow-Ten-Plus-Next-Digit-Into-Zero
0	1	$	Borrow-Treat-One-As-Zero
0	1		Borrow-Unit-Diff

		$	Borrow-Wont-Recurse
		$	Borrow-Wont-Recurse-Twice
			Borrowed-From-Don't-Borrow
1	0		Can't-Subtract
		$	Copy-Top-Except-Units
			Copy-Top-In-Last-Column-If-Borrowed-From
3	3		Decrement-All-On-Multiple-Zero
			Decrement-By-One-Plus-Zeros
			Decrement-By-Two-Over-Two
1	0		Decrement-Leftmost-Zero-Only
1	1		Decrement-Multiple-Zeros-By-Number-To-Left
3	1		Decrement-Multiple-Zeros-By-Number-To-Right
			Decrement-On-First-Borrow
0	1		Decrement-One-To-Eleven
			Decrement-One-To-Eleven-And-Continue
1	1		Decrement-Top-Leq-Is-Eight
0	13		Diff-0-N=0
1	42		Diff-0-N=N
0	2		Diff-0-N=N-When-Borrow-From-Zero
			Diff-1-N=1
0	6		Diff-N-0=0
0	1		Diff-N-N=N
0	1	$	Doesn't-Borrow-Except-Last
		$	Doesn't-Borrow-Unless-Bottom-Smaller
		$	Doesnt-Borrow
			Don't-Decrement-Second-Zero
3	4	$	Don't-Decrement-Zero
4	2		Don't-Decrement-Zero-Over-Blank
0	1		Don't-Decrement-Zero-Until-Bottom-Blank
1	3		Don't-Write-Zero
1	2		Double-Decrement-One
1	3	$	Forget-Borrow-Over-Blanks
0	6		Ignore-Leftmost-One-Over-Blank
			Ignore-Zero-Over-Blank
			Increment-Over-Larger
		$	Increment-Zero-Over-Blank
			Mix-Up-Six-And-Nine
			N-9=N-1-After-Borrow
0	2		N-N-After-Borrow-Causes-Borrow
1	0	$	N-N-Causes-Borrow
1	3		N-N=1-After-Borrow
			N-N=9-Plus-Decrement
			Once-Borrow-Always-Borrow
		$	*Only-Do-First&Last-Columns
0	1	$	Only-Do-Units
		$	*Only-Do-Units&Tens

		$	*Only-Do-Units-Unless-Two-Columns
0	5	$	Quit-When-Bottom-Blank
		$	*Quit-When-Second-Bottom-Blank
1	0		Simple-Problem-Stutter-Subtract
		$	*Skip-Interior-Bottom-Blank
103	12	$	Smaller-From-Larger
0	3	$	Smaller-From-Larger-Except-Last
0	5	$	Smaller-From-Larger-Instead-Of-Borrow-From-Zero
		$	Smaller-From-Larger-Insteadof-Borrow-From-Double-Zero
2	5	$	Smaller-From-Larger-Insteadof-Borrow-Unless-Bottom-Smaller
0	7		Smaller-From-Larger-When-Borrowed-From
		$	Smaller-From-Larger-With-Borrow
2	1	$	Stops-Borrow-At-Multiple-Zero
		$	Stops-Borrow-At-Second-Zero
34	30	$	Stops-Borrow-At-Zero
2	0		Stutter-Subtract
1	0		Sub-Bottom-From-Top
1	0		Sub-Copy-Least-Bottom-Most-Top
0	2		Sub-One-Over-Blank
		$	Top-After-Borrow
0	1	$	Top-Instead-Of-Borrow-From-Zero
			Top-Insteadof-Borrow
		$	Top-Insteadof-Borrow-Except-Last
		$	Top-Insteadof-Borrow-From-Double-Zero
		$	Top-Insteadof-Borrow-Unless-Bottom-Smaller
			Treat-Top-Zero-As-Nine
0	1		Treat-Top-Zero-As-Ten
0	1		X-N=0-After-Borrow
0	1		X-N=N-After-Borrow
			Zero-After-Borrow
			Zero-Instead-Of-Borrow-From-Zero
1	4		Zero-Insteadof-Borrow

Appendix 4
Sierra's predicted bug sets

This appendix lists all the bug sets generated by Sierra for the Southbay experiment. There are 119 of them.

(!Touched-0-N=0 Borrow-Across-Zero !Touched-Double-Zero-Is-Quit N-N-Causes-Borrow)
(!Touched-0-N=Blank Borrow-Across-Zero *Quit-When-Second-Bottom-Blank)
(!Touched-0-N=Blank Borrow-Across-Zero !Touched-Double-Zero-Is-Quit N-N-Causes-Borrow)
(!Touched-0-N=N Borrow-Across-Zero *Quit-When-Second-Bottom-Blank !Touched-Double-Zero-Is-Quit)
(!Touched-0-N=N Borrow-Across-Zero !Touched-Double-Zero-Is-Quit N-N-Causes-Borrow)
(!Touched-0-Is-Quit Borrow-Across-Zero Borrow-Wont-Recurse-Twice)
(!Touched-0-Is-Quit Borrow-Across-Zero *Quit-When-Second-Bottom-Blank)
(!Touched-0-Is-Quit Borrow-Across-Zero N-N-Causes-Borrow)
(!Touched-0-Is-Quit Borrow-Across-Zero *Skip-Interior-Bottom-Blank)
(!Touched-0-Is-Ten Borrow-Across-Zero Borrow-Wont-Recurse-Twice)
(!Touched-0-Is-Ten Borrow-Across-Zero *Quit-When-Second-Bottom-Blank !Touched-Double-Zero-Is-Quit)
(!Touched-0-Is-Ten Borrow-Across-Zero !Touched-Double-Zero-Is-Quit N-N-Causes-Borrow)
(Blank-Instead-Of-Borrow-From-Zero)
(Blank-Instead-Of-Borrow-From-Zero N-N-Causes-Borrow)
(Blank-Insteadof-Borrow *Only-Do-Units&Tens *Only-Do-Units-Unless-Two-Columns)
(Blank-Insteadof-Borrow *Only-Do-First&Last-Columns Quit-When-Bottom-Blank)
(Blank-Insteadof-Borrow *Only-Do-First&Last-Columns)
(Blank-Insteadof-Borrow-Except-Last)
(*Blank-Insteadof-Borrow-From-Double-Zero)
(Blank-Insteadof-Borrow-Unless-Bottom-Smaller)
(*Blank-With-Borrow Blank-Instead-Of-Borrow-From-Zero)
(*Blank-With-Borrow Borrow-Wont-Recurse)
(*Blank-With-Borrow Borrow-Add-Decrement-Insteadof-Zero)
(*Blank-With-Borrow Top-Instead-Of-Borrow-From-Zero)
(*Blank-With-Borrow Smaller-From-Larger-Instead-Of-Borrow-From-Zero)
(Borrow-Across-Second-Zero)
(Borrow-Across-Zero Borrow-Wont-Recurse-Twice)
(Borrow-Across-Zero !Touched-0-Is-Quit)
(Borrow-Across-Zero !Touched-Double-Zero-Is-Quit)
(Borrow-Across-Zero !Touched-0-Is-Ten !Touched-Double-Zero-Is-Quit)
(Borrow-Across-Zero !Touched-0-Is-Ten)
(Borrow-Across-Zero)
(Borrow-Across-Zero !Touched-0-N=Blank !Touched-Double-Zero-Is-Quit)
(Borrow-Across-Zero !Touched-0-N=0 !Touched-Double-Zero-Is-Quit)
(Borrow-Across-Zero !Touched-0-N=N !Touched-Double-Zero-Is-Quit)
(Borrow-Across-Zero *Quit-When-Second-Bottom-Blank !Touched-Double-Zero-Is-Quit)
(Borrow-Across-Zero !Touched-Double-Zero-Is-Quit N-N-Causes-Borrow)
(Borrow-Add-Decrement-Insteadof-Zero)
(Borrow-Add-Decrement-Insteadof-Zero *Quit-When-Second-Bottom-Blank)
(Borrow-Add-Decrement-Insteadof-Zero *Skip-Interior-Bottom-Blank)
(Borrow-Add-Decrement-Insteadof-Zero N-N-Causes-Borrow)
(Borrow-From-One-Is-Nine Borrow-From-Zero)
(Borrow-From-One-Is-Ten Borrow-From-Zero-Is-Ten)
(Borrow-From-Zero)
(Borrow-From-Zero-Is-Ten)
(Borrow-No-Decrement)

(Borrow-No-Decrement-Except-Last *Skip-Interior-Bottom-Blank)
(Borrow-No-Decrement-Except-Last)
(Borrow-No-Decrement-Except-Last *Quit-When-Second-Bottom-Blank)
(Borrow-Treat-One-As-Zero)
(Borrow-Wont-Recurse *Only-Do-Units-Unless-Two-Columns)
(Borrow-Wont-Recurse)
(Borrow-Wont-Recurse *Only-Do-First&Last-Columns)
(Borrow-Wont-Recurse *Quit-When-Second-Bottom-Blank)
(Borrow-Wont-Recurse *Skip-Interior-Bottom-Blank)
(Borrow-Wont-Recurse N-N-Causes-Borrow)
(Borrow-Wont-Recurse Smaller-From-Larger-With-Borrow)
(Borrow-Wont-Recurse-Twice)
(Doesn't-Borrow-Except-Last *Only-Do-Units-Unless-Two-Columns)
(Doesn't-Borrow-Except-Last *Only-Do-Units&Tens Doesn't-Borrow-Unless-Bottom-Smaller)
(Doesn't-Borrow-Except-Last *Only-Do-First&Last-Columns)
(Doesn't-Borrow-Except-Last *Only-Do-First&Last-Columns Copy-Top-Except-Units)
(Doesn't-Borrow-Except-Last)
(Doesn't-Borrow-Unless-Bottom-Smaller)
(Doesnt-Borrow)
(Doesnt-Borrow *Only-Do-Units&Tens)
(Doesnt-Borrow Blank-Instead-Of-Borrow-From-Zero)
(Doesnt-Borrow *Only-Do-First&Last-Columns Quit-When-Bottom-Blank)
(Doesnt-Borrow Smaller-From-Larger-Instead-Of-Borrow-From-Zero)
(Doesnt-Borrow *Only-Do-First&Last-Columns)
(Don't-Decrement-Zero)
(Forget-Borrow-Over-Blanks Borrow-Don't-Decrement-Unless-Bottom-Smaller)
(Increment-Zero-Over-Blank)
(*Only-Do-First&Last-Columns Borrow-No-Decrement-Except-Last)
(*Only-Do-First&Last-Columns)
(*Only-Do-First&Last-Columns Doesn't-Borrow-Except-Last)
(*Only-Do-First&Last-Columns Smaller-From-Larger Quit-When-Bottom-Blank)
(*Only-Do-First&Last-Columns Smaller-From-Larger)
(*Only-Do-First&Last-Columns Smaller-From-Larger-Except-Last)
(*Only-Do-First&Last-Columns Blank-Insteadof-Borrow-Except-Last)
(*Only-Do-First&Last-Columns Always-Borrow-Left)
(Only-Do-Units Blank-Insteadof-Borrow)
(*Only-Do-Units&Tens Blank-Insteadof-Borrow)
(*Only-Do-Units&Tens *Only-Do-Units-Unless-Two-Columns Smaller-From-Larger-Except-Last)
(*Only-Do-Units-Unless-Two-Columns)
(*Only-Do-Units-Unless-Two-Columns Copy-Top-Except-Units)
(*Quit-When-Second-Bottom-Blank Smaller-From-Larger-Instead-Of-Borrow-From-Zero)
(*Quit-When-Second-Bottom-Blank Blank-Instead-Of-Borrow-From-Zero)
(*Quit-When-Second-Bottom-Blank)
(*Quit-When-Second-Bottom-Blank Stops-Borrow-At-Zero)
(*Skip-Interior-Bottom-Blank Doesn't-Borrow-Except-Last)
(*Skip-Interior-Bottom-Blank Smaller-From-Larger-Except-Last)
(*Skip-Interior-Bottom-Blank Blank-Insteadof-Borrow-Except-Last)
(*Skip-Interior-Bottom-Blank Smaller-From-Larger-Instead-Of-Borrow-From-Zero)
(*Skip-Interior-Bottom-Blank Blank-Instead-Of-Borrow-From-Zero)
(*Skip-Interior-Bottom-Blank Stops-Borrow-At-Zero)
(Smaller-From-Larger *Only-Do-Units-Unless-Two-Columns Quit-When-Bottom-Blank)
(Smaller-From-Larger-Except-Last)
(Smaller-From-Larger-Instead-Of-Borrow-From-Zero)
(Smaller-From-Larger-Instead-Of-Borrow-From-Zero N-N-Causes-Borrow)
(Smaller-From-Larger-Insteadof-Borrow-From-Double-Zero)
(Smaller-From-Larger-Insteadof-Borrow-Unless-Bottom-Smaller)
(Smaller-From-Larger-With-Borrow Smaller-From-Larger-Instead-Of-Borrow-From-Zero)
(Smaller-From-Larger-With-Borrow Blank-Instead-Of-Borrow-From-Zero)
(Smaller-From-Larger-With-Borrow Borrow-Add-Decrement-Insteadof-Zero)

(Smaller-From-Larger-With-Borrow Top-Instead-Of-Borrow-From-Zero)
(Stops-Borrow-At-Multiple-Zero)
(Stops-Borrow-At-Second-Zero)
(Stops-Borrow-At-Zero)
(Stops-Borrow-At-Zero N-N-Causes-Borrow)
(Top-After-Borrow Stops-Borrow-At-Zero)
(Top-After-Borrow Borrow-Wont-Recurse)
(Top-After-Borrow Borrow-Add-Decrement-Insteadof-Zero)
(Top-Instead-Of-Borrow-From-Zero)
(Top-Instead-Of-Borrow-From-Zero N-N-Causes-Borrow)
(Top-Insteadof-Borrow-Except-Last)
(Top-Insteadof-Borrow-From-Double-Zero)
(Top-Insteadof-Borrow-Unless-Bottom-Smaller)

Appendix 5

Observed bug sets, overlapped by predicted bug sets

This takes the 134 bug sets that occurred in the Southbay data and lists them in three groups. The first group contains 11 bug sets that are identical to some bug set in the set of Sierra's predicted bug sets. The second group contains 47 bug sets that contain only bugs that Sierra cannot generate. The third group contains 76 bug sets that have non-empty intersections with at least one bug set from Sierra's predictions. This third group of bug sets is printed a little differently. The intersection is on one line, surrounded by parentheses, and the rest of the bug set, if any, is on the next line. Thus, if {A B C} and {C} are observed bug sets, and {C D} is in a predicted bug set, then the observed bug sets will be printed as:

```
((C)
  (A B))
((C)
  )
```

The other bug sets are printed in the usual way, as parenthesized lists.

Observed bug sets that are predicted by Sierra

34 occurrences
(Stops-Borrow-At-Zero)

13 occurrences
(Borrow-Across-Zero)

10 occurrences
(Borrow-From-Zero)
(Borrow-No-Decrement)

6 occurrences
(Borrow-Across-Zero !Touched-0-Is-Ten)

4 occurrences
(Borrow-No-Decrement-Except-Last)

3 occurrences
(Don't-Decrement-Zero)

2 occurrences
(Borrow-Across-Second-Zero)
(Smaller-From-Larger-Insteadof-Borrow-Unless-Bottom-Smaller)
(Stops-Borrow-At-Multiple-Zero)

1 occurrence
(Borrow-From-Zero-Is-Ten)

Observed bug sets that have no predicted bugs in them

6 occurrences
(Borrow-Across-Zero-Over-Zero Borrow-Across-Zero-Over-Blank)

4 occurrences
(0-N=N-Except-After-Borrow)
(Diff-0-N=N Zero-Insteadof-Borrow)
(Don't-Decrement-Zero-Over-Blank)

3 occurrences
(Decrement-All-On-Multiple-Zero)
(Decrement-Multiple-Zeros-By-Number-To-Right)

2 occurrences
(0-N=0-After-Borrow)
(Borrow-Across-Top-Smaller-Decrementing-To)
(Borrow-Don't-Decrement-Top-Smaller)
(Borrow-Only-From-Top-Smaller Borrow-Across-Zero-Over-Zero Borrow-Across-Zero-Over-Blank)
(Stutter-Subtract)

1 occurrence
(!Only-Write-Units-Digit N-N-After-Borrow-Causes-Borrow)
(0-N=N-After-Borrow Borrow-Across-Zero-Over-Zero Borrow-Across-Zero-Over-Blank)
(0-N=N-After-Borrow)
(0-N=N-Except-After-Borrow 1-1=0-After-Borrow)
(0-N=N-Except-After-Borrow 1-1=1-After-Borrow)
(1-1=0-After-Borrow)
(Add-Insteadof-Sub)
(Add-Lr-Decrement-Answer-Carry-To-Right}
(Borrow-Across-Zero-Over-Zero 0-N=N-Except-After-Borrow 1-1=0-After-Borrow)
(Borrow-Across-Zero-Over-Zero)
(Borrow-From-All-Zero)
(Borrow-From-Bottom-Insteadof-Zero Diff-0-N=N)
(Borrow-From-Zero&Left-Ten-Ok 0-N=N-After-Borrow)
(Borrow-From-Zero&Left-Ten-Ok)
(Borrow-Into-One=Ten Decrement-Multiple-Zeros-By-Number-To-Left)
(Borrow-Into-One=Ten Decrement-Multiple-Zeros-By-Number-To-Right Borrow-Across-Zero-Over-Zero)
(Borrow-Only-From-Top-Smaller)
(Borrow-Only-From-Top-Smaller 0-N=N-After-Borrow)
(Can't-Subtract)
(Decrement-All-On-Multiple-Zero Double-Decrement-One)
(Decrement-Leftmost-Zero-Only)
(Decrement-Multiple-Zeros-By-Number-To-Left)
(Decrement-Top-Leq-Is-Eight)
(Diff-0-N=N)
(Diff-0-N=N Diff-N-0=0)
(Don't-Decrement-Zero-Until-Bottom-Blank Borrow-Across-Zero-Over-Zero)
(Don't-Write-Zero)
(Double-Decrement-One)
(Double-Decrement-One Smaller-From-Larger-When-Borrowed-From)
(Ignore-Leftmost-One-Over-Blank Decrement-All-On-Multiple-Zero)
(N-N=1-After-Borrow 0-N=N-Except-After-Borrow)
(N-N=1-After-Borrow)
(Simple-Problem-Stutter-Subtract)
(Sub-Bottom-From-Top)
(Sub-Copy-Least-Bottom-Most-Top)
(Zero-Insteadof-Borrow)

Observed bug sets that overlap some predicted bug set

103 occurrences
((Smaller-From-Larger)
)

7 occurrences
((Stops-Borrow-At-Zero)
Diff-0-N=N)

6 occurrences
((Always-Borrow-Left)
)
((Borrow-Across-Zero)
Diff-0-N=N)
((Stops-Borrow-At-Zero)
Borrow-Once-Then-Smaller-From-Larger Diff-0-N=N)

5 occurrences
((Borrow-No-Decrement)
Diff-0-N=N)

4 occurrences
((Borrow-Across-Zero)
Diff-0-N=0)
((Quit-When-Bottom-Blank Smaller-From-Larger)
)

3 occurrences
((Stops-Borrow-At-Zero)
Borrow-Into-One=Ten)
((Smaller-From-Larger)
Ignore-Leftmost-One-Over-Blank)

2 occurrences
((Borrow-Don't-Decrement-Unless-Bottom-Smaller)
)
((Smaller-From-Larger-Instead-Of-Borrow-From-Zero)
Borrow-Once-Then-Smaller-From-Larger Diff-0-N=N)
((Stops-Borrow-At-Zero)
Smaller-From-Larger-When-Borrowed-From)
((Stops-Borrow-At-Zero)
Diff-0-N=N Smaller-From-Larger-When-Borrowed-From)

1 occurrence
((Stops-Borrow-At-Multiple-Zero)
!Only-Write-Units-Digit N-N-After-Borrow-Causes-Borrow)
((Borrow-Across-Zero)
!Sub-Units-Special Smaller-From-Larger)
((Smaller-From-Larger)
!Write-Left-Ten Diff-0-N=0)
((Borrow-Across-Second-Zero)
!Write-Left-Ten Diff-0-N=N)
((Forget-Borrow-Over-Blanks)
!Write-Left-Ten Diff-0-N=N)
((!Touched-0-N=N Borrow-Across-Zero)
Diff-N-0=0)
((!Touched-0-N=N Borrow-Across-Zero)

```
  Borrow-Once-Then-Smaller-From-Larger)
((!Touched-0-N=N  Borrow-Across-Zero)
  Borrow-Across-Second-Zero  Smaller-From-Larger-When-Borrowed-From)
((Smaller-From-Larger-Instead-Of-Borrow-From-Zero)
  0-N=N-After-Borrow  N-N=1-After-Borrow)
((Blank-Insteadof-Borrow)
  Diff-0-N=N)
((Borrow-Across-Second-Zero)
  Don't-Write-Zero)
((Borrow-Across-Second-Zero)
  Borrow-Skip-Equal)
((Borrow-Across-Zero)
  0-N=0-Except-After-Borrow)
((Borrow-Across-Zero)
  1-1=0-After-Borrow)
((Borrow-Across-Zero)
  Borrow-Once-Then-Smaller-From-Larger  0-N=N-Except-After-Borrow)
((Borrow-Across-Zero)
  Sub-One-Over-Blank  0-N=N-After-Borrow  0-N=N-Except-After-Borrow)
((Borrow-Across-Zero)
  Borrow-Skip-Equal)
((Borrow-Across-Zero)
  Quit-When-Bottom-Blank  0-N=0-After-Borrow)
((Borrow-Across-Zero)
  Forget-Borrow-Over-Blanks  Diff-0-N=N)
((Borrow-Across-Zero)
  Ignore-Leftmost-One-Over-Blank  Borrow-Skip-Equal)
((Borrow-Across-Zero  !Touched-0-N=N)
)
((Borrow-Don't-Decrement-Unless-Bottom-Smaller)
  X-N=0-After-Borrow)
((Borrow-Don't-Decrement-Unless-Bottom-Smaller)
  Don't-Write-Zero)
((Borrow-From-One-Is-Nine  Borrow-From-Zero)
  Diff-0-N=N-When-Borrow-From-Zero)
((Borrow-From-One-Is-Nine  Borrow-From-Zero)
  Don't-Decrement-Zero-Over-Blank)
((Borrow-From-One-Is-Ten  Borrow-From-Zero-Is-Ten)
  Borrow-Only-Once)
((Borrow-From-Zero)
  0-N=0-After-Borrow)
((Borrow-From-Zero)
  0-N=N-After-Borrow)
((Borrow-No-Decrement)
  Diff-0-N=0)
((Borrow-No-Decrement)
  Smaller-From-Larger-Except-Last  Smaller-From-Larger-Insteadof-Borrow-Unless-Bottom-Smaller
Diff-0-N=N)
((Borrow-No-Decrement)
  Sub-One-Over-Blank)
((Borrow-No-Decrement-Except-Last)
  Treat-Top-Zero-As-Ten)
((Borrow-No-Decrement-Except-Last)
  Decrement-Top-Leq-Is-Eight  X-N=N-After-Borrow)
((Borrow-Treat-One-As-Zero)
  N-N=1-After-Borrow  Don't-Decrement-Zero-Over-Blank)
((Only-Do-Units)
  Borrow-Unit-Diff)
((Stops-Borrow-At-Zero)
  Diff-0-N=0  Diff-N-0=0)
((Doesn't-Borrow-Except-Last)
  Diff-0-N=0  Diff-N-0=0  Smaller-From-Larger-Insteadof-Borrow-Unless-Bottom-Smaller)
((Smaller-From-Larger-Except-Last)
  Diff-0-N=0  Diff-N-0=0  Smaller-From-Larger-Insteadof-Borrow-Unless-Bottom-Smaller)
((Don't-Decrement-Zero)
  Diff-0-N=N-When-Borrow-From-Zero)
((Smaller-From-Larger)
  Diff-N-0=0  Diff-0-N=0)
((Borrow-Across-Second-Zero)
  Don't-Decrement-Zero)
((Don't-Decrement-Zero)
  1-1=0-After-Borrow)
((Don't-Decrement-Zero)
  Decrement-One-To-Eleven)
((Forget-Borrow-Over-Blanks)
)
((Forget-Borrow-Over-Blanks)
  Borrow-Don't-Decrement-Top-Smaller  Borrow-Skip-Equal)
((N-N-Causes-Borrow)
)
((Smaller-From-Larger)
  Diff-N-N=N  Diff-0-N=0)
((Smaller-From-Larger)
  Diff-0-N=0)
((Smaller-From-Larger-Except-Last)
  Decrement-All-On-Multiple-Zero)
((Smaller-From-Larger-Instead-Of-Borrow-From-Zero)
  Diff-0-N=N  Smaller-From-Larger-When-Borrowed-From)
((Smaller-From-Larger-Instead-Of-Borrow-From-Zero)
  Borrow-Once-Then-Smaller-From-Larger)
((Smaller-From-Larger-Insteadof-Borrow-Unless-Bottom-Smaller)
  0-N=N-Except-After-Borrow)
((Top-Instead-Of-Borrow-From-Zero)
  Smaller-From-Larger-Insteadof-Borrow-Unless-Bottom-Smaller  Diff-0-N=N)
((Stops-Borrow-At-Zero)
  0-N=0-Except-After-Borrow  1-1=0-After-Borrow)
((Stops-Borrow-At-Zero)
  Borrow-Across-Zero-Over-Zero  1-1=0-After-Borrow)
((Stops-Borrow-At-Zero)
  1-1=1-After-Borrow)
((Stops-Borrow-At-Zero)
  Diff-0-N=0)
((Stops-Borrow-At-Zero)
  Ignore-Leftmost-One-Over-Blank)
((Stops-Borrow-At-Zero)
  0-N=0-After-Borrow)
((Stops-Borrow-At-Zero)
  Borrow-Once-Then-Smaller-From-Larger)
((Stops-Borrow-At-Zero)
  1-1=0-After-Borrow)
((Stops-Borrow-At-Zero)
  Diff-0-N=N  Don't-Write-Zero)
```

Appendix 6
Predicted bug sets, overlapped by observed bug sets

This takes the 119 bug sets that were generated by Sierra for the Southbay experiment and lists them in three groups. The first group contains 11 bug sets that are identical to some observed bug set. The second group contains 40 bug sets that contain only unobserved bugs. The third group contains 68 bug sets that have non-empty intersections when interesected with at least one observed bug set. This third group of bug sets is printed a little differently. The intersection is on one line, surrounded by parentheses, and the rest of the bug set, if any, is on the next line. Thus, if {A B C} and {C} are predicted bug set, and {C D} is an observed bug set, then the two predicted bug set will be printed as:

```
((C)
  (A B))
((C)
  )
```

The other bug sets are printed in the usual way, as parenthesized lists.

Predicted bug sets, identical to some observed bug set

(Stops-Borrow-At-Zero)
(Borrow-Across-Zero)
(Borrow-From-Zero)
(Borrow-No-Decrement)
(Borrow-Across-Zero !Touched-0-Is-Ten)
(Borrow-No-Decrement-Except-Last)
(Don't-Decrement-Zero)
(Borrow-Across-Second-Zero)
(Smaller-From-Larger-Insteadof-Borrow-Unless-Bottom-Smaller)
(Stops-Borrow-At-Multiple-Zero)
(Borrow-From-Zero-Is-Ten)

Predicted bug sets with no observed bugs

(Blank-Instead-Of-Borrow-From-Zero)
(Blank-Insteadof-Borrow-Except-Last)
(*Blank-Insteadof-Borrow-From-Double-Zero)
(Blank-Insteadof-Borrow-Unless-Bottom-Smaller)
(*Blank-With-Borrow Blank-Instead-Of-Borrow-From-Zero)
(*Blank-With-Borrow Borrow-Wont-Recurse)
(*Blank-With-Borrow Borrow-Add-Decrement-Insteadof-Zero)
(Borrow-Add-Decrement-Insteadof-Zero)
(Borrow-Add-Decrement-Insteadof-Zero *Quit-When-Second-Bottom-Blank)
(Borrow-Add-Decrement-Insteadof-Zero *Skip-Interior-Bottom-Blank)
(Borrow-Wont-Recurse *Only-Do-Units-Unless-Two-Columns)
(Borrow-Wont-Recurse)
(Borrow-Wont-Recurse *Only-Do-First&Last-Columns)
(Borrow-Wont-Recurse *Quit-When-Second-Bottom-Blank)
(Borrow-Wont-Recurse *Skip-Interior-Bottom-Blank)
(Borrow-Wont-Recurse Smaller-From-Larger-With-Borrow)
(Borrow-Wont-Recurse-Twice)
(Doesn't-Borrow-Unless-Bottom-Smaller)

```
(Doesnt-Borrow)
(Doesnt-Borrow  *Only-Do-Units&Tens)
(Doesnt-Borrow  Blank-Instead-Of-Borrow-From-Zero)
(Doesnt-Borrow  *Only-Do-First&Last-Columns)
(Increment-Zero-Over-Blank)
(*Only-Do-First&Last-Columns)
(*Only-Do-First&Last-Columns  Blank-Insteadof-Borrow-Except-Last)
(*Only-Do-Units-Unless-Two-Columns)
(*Only-Do-Units-Unless-Two-Columns  Copy-Top-Except-Units)
(*Quit-When-Second-Bottom-Blank  Blank-Instead-Of-Borrow-From-Zero)
(*Quit-When-Second-Bottom-Blank)
(*Skip-Interior-Bottom-Blank  Blank-Insteadof-Borrow-Except-Last)
(*Skip-Interior-Bottom-Blank  Blank-Instead-Of-Borrow-From-Zero)
(Smaller-From-Larger-Insteadof-Borrow-From-Double-Zero)
(Smaller-From-Larger-With-Borrow  Blank-Instead-Of-Borrow-From-Zero)
(Smaller-From-Larger-With-Borrow  Borrow-Add-Decrement-Insteadof-Zero)
(Stops-Borrow-At-Second-Zero)
(Top-After-Borrow  Borrow-Wont-Recurse)
(Top-After-Borrow  Borrow-Add-Decrement-Insteadof-Zero)
(Top-Insteadof-Borrow-Except-Last)
(Top-Insteadof-Borrow-From-Double-Zero)
(Top-Insteadof-Borrow-Unless-Bottom-Smaller)
```

Predicted bug sets, overlapped by observed bug sets

```
((Borrow-Across-Zero)
  !Touched-0-N=0  !Touched-Double-Zero-Is-Quit  N-N-Causes-Borrow)
((Borrow-Across-Zero)
  !Touched-0-N=Blank  *Quit-When-Second-Bottom-Blank)
((Borrow-Across-Zero)
  !Touched-0-N=Blank  !Touched-Double-Zero-Is-Quit  N-N-Causes-Borrow)
((!Touched-0-N=N  Borrow-Across-Zero)
  *Quit-When-Second-Bottom-Blank  !Touched-Double-Zero-Is-Quit)
((!Touched-0-N=N  Borrow-Across-Zero)
  !Touched-Double-Zero-Is-Quit  N-N-Causes-Borrow)
((Borrow-Across-Zero)
  !Touched-0-Is-Quit  Borrow-Wont-Recurse-Twice)
((Borrow-Across-Zero)
  !Touched-0-Is-Quit  *Quit-When-Second-Bottom-Blank)
((Borrow-Across-Zero)
  !Touched-0-Is-Quit  N-N-Causes-Borrow)
((Borrow-Across-Zero)
  !Touched-0-Is-Quit  *Skip-Interior-Bottom-Blank)
((Borrow-Across-Zero)
  !Touched-0-Is-Ten  Borrow-Wont-Recurse-Twice)
((Borrow-Across-Zero)
  !Touched-0-Is-Ten  *Quit-When-Second-Bottom-Blank  !Touched-Double-Zero-Is-Quit)
((Borrow-Across-Zero)
  !Touched-0-Is-Ten  !Touched-Double-Zero-Is-Quit  N-N-Causes-Borrow)
((N-N-Causes-Borrow)
  Blank-Instead-Of-Borrow-From-Zero)
((Blank-Insteadof-Borrow)
  *Only-Do-Units&Tens  *Only-Do-Units-Unless-Two-Columns)
((Quit-When-Bottom-Blank)
  Blank-Insteadof-Borrow  *Only-Do-First&Last-Columns)
((Blank-Insteadof-Borrow)
  *Only-Do-First&Last-Columns)
((Top-Instead-Of-Borrow-From-Zero)
  *Blank-With-Borrow)
((Smaller-From-Larger-Instead-Of-Borrow-From-Zero)
  *Blank-With-Borrow)
((Borrow-Across-Zero)
  Borrow-Wont-Recurse-Twice)
((Borrow-Across-Zero)
  !Touched-0-Is-Quit)
((Borrow-Across-Zero)
  !Touched-Double-Zero-Is-Quit)
((Borrow-Across-Zero)
  !Touched-0-Is-Ten  !Touched-Double-Zero-Is-Quit)
((Borrow-Across-Zero)
  !Touched-0-N = Blank  !Touched-Double-Zero-Is-Quit)
((Borrow-Across-Zero)
  !Touched-0-N = 0  !Touched-Double-Zero-Is-Quit)
((Borrow-Across-Zero  !Touched-0-N = N)
  !Touched-Double-Zero-Is-Quit)
((Borrow-Across-Zero)
  *Quit-When-Second-Bottom-Blank  !Touched-Double-Zero-Is-Quit)
((Borrow-Across-Zero)
  !Touched-Double-Zero-Is-Quit  N-N-Causes-Borrow)
((N-N-Causes-Borrow)
  Borrow-Add-Decrement-Insteadof-Zero)
((Borrow-From-One-Is-Nine  Borrow-From-Zero)
)
((Borrow-From-One-Is-Ten  Borrow-From-Zero-Is-Ten)
)
((Borrow-No-Decrement-Except-Last)
  *Skip-Interior-Bottom-Blank)
((Borrow-No-Decrement-Except-Last)
  *Quit-When-Second-Bottom-Blank)
((Borrow-Treat-One-As-Zero)
)
((N-N-Causes-Borrow)
  Borrow-Wont-Recurse)
((Doesn't-Borrow-Except-Last)
  *Only-Do-Units-Unless-Two-Columns)
((Doesn't-Borrow-Except-Last)
  *Only-Do-Units&Tens  Doesn't-Borrow-Unless-Bottom-Smaller)
((Doesn't-Borrow-Except-Last)
  *Only-Do-First&Last-Columns)
((Doesn't-Borrow-Except-Last)
  *Only-Do-First&Last-Columns  Copy-Top-Except-Units)
((Doesn't-Borrow-Except-Last)
)
((Quit-When-Bottom-Blank)
  Doesnt-Borrow  *Only-Do-First&Last-Columns)
((Smaller-From-Larger-Instead-Of-Borrow-From-Zero)
  Doesnt-Borrow)
((Borrow-Don't-Decrement-Unless-Bottom-Smaller)
  Forget-Borrow-Over-Blanks)
((Borrow-No-Decrement-Except-Last)
  *Only-Do-First&Last-Columns)
((Doesn't-Borrow-Except-Last)
  *Only-Do-First&Last-Columns)
((Smaller-From-Larger  Quit-When-Bottom-Blank)
  *Only-Do-First&Last-Columns)
((Smaller-From-Larger)
  *Only-Do-First&Last-Columns)
```

```
((Smaller-From-Larger-Except-Last)
  *Only-Do-First&Last-Columns)
((Always-Borrow-Left)
  *Only-Do-First&Last-Columns)
((Blank-Insteadof-Borrow)
  Only-Do-Units)
((Blank-Insteadof-Borrow)
  *Only-Do-Units&Tens)
((Smaller-From-Larger-Except-Last)
  *Only-Do-Units&Tens  *Only-Do-Units-Unless-Two-Columns)
((Smaller-From-Larger-Instead-Of-Borrow-From-Zero)
  *Quit-When-Second-Bottom-Blank)
((Stops-Borrow-At-Zero)
  *Quit-When-Second-Bottom-Blank)
((Doesn't-Borrow-Except-Last)
  *Skip-Interior-Bottom-Blank)
((Smaller-From-Larger-Except-Last)
  *Skip-Interior-Bottom-Blank)
((Smaller-From-Larger-Instead-Of-Borrow-From-Zero)
  *Skip-Interior-Bottom-Blank)
((Stops-Borrow-At-Zero)
  *Skip-Interior-Bottom-Blank)
((Smaller-From-Larger  Quit-When-Bottom-Blank)
  *Only-Do-Units-Unless-Two-Columns)
((Smaller-From-Larger)
  *Only-Do-Units&Tens  0-N=N-Except-After-Borrow)
((Smaller-From-Larger-Except-Last)
)
((Smaller-From-Larger-Instead-Of-Borrow-From-Zero)
)
((Smaller-From-Larger-Instead-Of-Borrow-From-Zero)
  N-N-Causes-Borrow)
((Smaller-From-Larger-Instead-Of-Borrow-From-Zero)
  Smaller-From-Larger-With-Borrow)
((Top-Instead-Of-Borrow-From-Zero)
  Smaller-From-Larger-With-Borrow)
((Stops-Borrow-At-Zero)
  N-N-Causes-Borrow)
((Stops-Borrow-At-Zero)
  Top-After-Borrow)
((Top-Instead-Of-Borrow-From-Zero)
)
((N-N-Causes-Borrow)
  Top-Instead-Of-Borrow-From-Zero)
```

Appendix 7

Version Spaces as Applicability Conditions

Chapter 19 discussed what the learner's bias is for test patterns. It was shown that a topological bias — preferring maximally general test patterns — was better than a bias based on teleological rationalizations. However, there is a problem with this choice. This appendix discusses that problem, and proposes a revision to the theory that fixes it. However, this revision has not (yet) been tried out on Sierra. The arguments should be taken with a grain of salt.

The topological bias depends crucially on negative instances

The topological bias takes the most general test patterns that are consistent with the examples. If there are no negative instances, the most general test pattern is the trivially general pattern, which matches all possible problem states. Thus, if a lesson has no descrimination examples (i.e., examples with negative instances of the subprocedure's test pattern), the new subprocedure will have a test pattern that is always true. Such a lesson generates the bug

Always-Borrow:

$$\begin{array}{r} {}^{4}\phantom{^{1}7} \\ 5^{1}7 \\ -\ 1\ 2 \\ \hline 315 \end{array}\ \times \qquad \begin{array}{r} {}^{4}\ {}^{1}{}_{5} \\ 5\ 6^{1}0 \\ -\ 2\ 8\ 0 \\ \hline 2\ 710 \end{array}\ \times$$

This bug borrows in every column, regardless of the relative values of the column's digits. (N.B., It tries to borrow in the leftmost column, does some local problem solving, and winds up doing an ordinary column difference in the leftmost column.) This bug is predicted for lessons that teach borrow without giving descrimination examples. For borrowing, one descrimination example is *a*:

$$\text{a.}\quad \begin{array}{r} 5\ 7 \\ -\ 2\ 1 \\ \hline 3\ 6 \end{array} \qquad \text{b.}\quad \begin{array}{r} 5\ 7 \\ -\ 2\ 1 \\ \hline \end{array} \qquad \text{c.}\quad \begin{array}{r} 5\ 3 \\ -\ 1\ 7 \\ \hline \end{array}$$

At the beginning of the problem's solution (state *b*), when the goal `SUB1COL` is called to process the units column, it has a choice between borrowing and taking the usual column difference. Because the teacher does an ordinary column difference, the learner can infer that the teacher's test pattern for `BORROW` was false. (This inference involves the conflict resolution conventions of the interpreter — see section 10.5.) The inferred falsehood of the teacher's test pattern means that current problem state, *b*, is a negative instance. The learner compares it to generalizations of positive instances, such as the problem state *c*, where the test pattern was true. The only difference between the negative instance and the generalized positive instance is that T<B is false in the negative instance's units column. Hence, T<B is a most general generalization: If the test pattern were only T<B, then it would be consistent with all the instances, including the negative instance.

What if there are no discrimination examples in a lesson?

Some textbooks do not have discrimination examples in their lessons. Houghton Mifflin's 1981 textbook series, for example, does not use discrimination examples in teaching subtraction. The absence of discrimination examples would seem to predict that all Houghton Mifflin students acquire the bug Always-Borrow. But surely some students learn the correct subtraction procedure.

The apparent contradiction stems from the tacit assumption that a negative instance has to be in a subprocedure's lesson in order for it to be effective. This assumption is apparently false. In fact, the lessons following the initial borrow lessons in the Houghton Mifflin text have the requisite negative instances. In a borrow-from-zero example, there is always a problem state which is a negative instance for borrowing (e.g., *d* below).

d.

$$\begin{array}{r} 5\ 9\ \ \\ 6^{1}0^{1}7 \\ -\ 2\ 3\ 8 \\ \hline 9 \end{array}$$

In order for Sierra's learner to make use of negative instances, it must remember something about the positive instances. Induction needs to know what was true about the positive instances so that it can determine which of these truths is false of the negative instances. Some induction algorithms store all the positive instances (c.f., Dietterich & Michalski, 1981). The version space algorithm (see section 18.1) saves the maximally specific generalizations of the positive instances. In order for the learner to make use of the negative borrow instances in, e.g., the borrow-from-zero lesson, it must store information about positive borrow instances from the borrow lesson. This information can not reside in the test pattern. Since there have been no negative instances, the test pattern is trivially general, i.e., {}. In order to save the information, there must be ancillary information storage in the student procedure.

Using version spaces as applicability conditions

The obvious solution is to have the procedure store the whole version space for a test pattern instead of just the test pattern itself. This technique has been used before, for different reasons, by Mitchell et. al. (1981; 1983) in their calculus learner, LEX. LEX uses version spaces as tests. Given a version space ⟨S, G⟩, the interpretation of it as a test is:

A. A test is true if for all $s \in S$, s matches the current problem state.

B. A test is false if there is no $g \in G$ such that g matches the current problem state.

C. If a test is neither true nor false, its value is called *indeterminate.*

This interpretation is logically correct, in a sense. Since the $g \in G$ are all maximally general generalizations, the target test pattern, whatever it is, is less general than them. Hence, if none of the g match, then the test pattern couldn't match. So if no g matches, one can infer that the test pattern would be false. Similarly, if all the maximally specific generalizations match, then the unknown test pattern would also match since it is a generalization of at least one of them. If some g match and some s don't match, then one can infer nothing about the truth of the unknown target pattern. Depending on where the target pattern lies between the s and the g patterns, it could be either true or false. So a third truth value is needed. It is labelled indeterminate.

A version space is represented by two *sets* of patterns, S and G. There is ample evidence that rules use only a single pattern of each kind. Basically, if this were not the case, then every learner would learn exactly the same applicability conditions since, by definition, only one version space results from a given set of examples. All the overgeneralization bugs would be ruled out if the whole version space were used as the applicability condition. In short, to account for the fact that different learners induce different applicability conditions, version spaces must be used in the following way:

Version space
If <S, G> is the version space for an applicability condition, then for all $s \in G$ and for all $g \in G$, **Induce** outputs an adjoining rule with the pair <*s*, *g*> as the applicability condition. The condition is true if *s* matches exactly, false if *g* doesn't match exactly, and indeterminate otherwise.

This hypothesis replaces two hypotheses in the theory: test pattern bias and test pattern matching. Moreover, a third hypothesis must be revised slightly. The interpreter must have some conventions for handling the indeterminate truth value. The conflict resolution hypothesis must be revised to become:

Conflict resolution

1. A rule may be executed only if it has not yet been executed for this instance of the current goal.
2. In the representation of procedures, the rules of AND goals are linearly ordered. If the current goal is an AND goal, and there are several unexecuted applicable rules, then the interpreter picks the first one in the goal's order.
3. If the current goal is an OR goal, then
   - A. If one rule's test is true, then the interpreter picks that rule.
   - B. If more than one rule's test is true, the interpreter picks the rule that was acquired most recently.
   - C. If no rule's test is true, and exactly one rule's test is indeterminate (the others being false), the interpreter picks the rule with the indeterminate test.
   - D. Otherwise, a halt impasse occurs.

Clause B is a common conflict resolution strategy in production systems (McDermott & Forgy, 1978). Here it is used because it is necessary in order to allow the learner to infer negative instances for test pattern induction (see section 10.5). Clauses A and C are plain common sense: Take a true rule if there is just one and don't take false rules.

More direct arguments for using version spaces as applicability conditions

The above argument for representing test patterns as version spaces was based on two rather dubious criteria. First, it assumed that the learning model's chronology should mimic the chronology of human learners at the grain size of lessons. Second, it assumed that all information from a lesson is transmitted chronologically via the learner's procedure. Both criteria go beyond the main empirical criterion, which involves predicting the observed bugs. So the argument above is more an informal motivation than a theoretically impeccable demonstration. Nonetheless, its conclusion makes some verifiable bug predictions. These provide a proper support for the position that applicability conditions use version spaces.

The conflict resolution conventions cause impasses in exactly the right places to generate certain bugs, such as the following one:

Smaller-From-Larger-When-Borrowed-From:

a. $$\begin{array}{r} 4\phantom{{}^{1}7\;6} \\ \cancel{5}{}^{1}7\;6 \\ -\;1\;9\;1 \\ \hline 3\;8\;5 \end{array}\ \checkmark$$

b. $$\begin{array}{r} 1\phantom{{}^{1}7} \\ 5\;2^{1}7 \\ -\;1\;8\;9 \\ \hline 4\;7\;8 \end{array}\ \times$$

The learner can do problems with isolated borrows, as in *a*, but when confronted with a problem that requires adjacent borrows, such as *b*, the solver impasses instead of doing the second borrow. This bug derives from the local problem solver taking the Force repair (which will be explained in a moment). This repair ultimately results in the solver answering the tens column of *b* by inverting it (i.e., the answer is 8−1). The important issue is the cause of the impasse. It seems that this bug occurs when the learner has not yet taken the lesson on adjacent borrows. That is, the learner assimilated examples such as *c*, *d* and *e* below, but not ones like *f*.

c. $$\begin{array}{r} 4\phantom{{}^{1}7} \\ \cancel{5}{}^{1}7 \\ -\;1\;9 \\ \hline 3\;8 \end{array}$$

d. $$\begin{array}{r} 4\phantom{{}^{1}7\;2} \\ \cancel{5}{}^{1}7\;2 \\ -\;1\;9\;1 \\ \hline 1\;8\;1 \end{array}$$

e. $$\begin{array}{r} 4\phantom{{}^{1}7} \\ 2\;\cancel{5}{}^{1}7 \\ -\;1\;1\;9 \\ \hline 1\;3\;8 \end{array}$$

f. $$\begin{array}{r} 2\;{}^{1}4\phantom{{}^{1}7} \\ 3\;\cancel{5}{}^{1}7 \\ -\;1\;6\;9 \\ \hline 1\;8\;8 \end{array}$$

The presence of negative instances in *d* and *e* means that the version space for borrow's test will have non-trivial maximally general generalizations. In fact, the borrow rule's version space will be roughly:

```
G:   {{(LessThan? T B)}}
S:   {{...(LessThan? T B)(DIGIT T)...} ...}
```

There is only one maximally general generalization, T<B. Crucially, some of the maximally specific generalizations in S expect the top member of the column to be a `DIGIT`, not a crossed-out digit (which is categorized as an `XNUM` by the grammar). They expect this because in all the positive instances (e.g., in *c*, *d* and *e*), the top cell of the column isn't crossed out. The first time the inducer can encounter a crossed-out top digit in a column that requires borrowing is in problems like *f* that require adjacent borrows. The learner hasn't encountered such problems yet. At least one $s \in S$ (if not all) expect `(DIGIT T)`. In problem *b*, the top digit of the tens column is crossed-out when borrow's version space is tested. Hence, at least one $s \in S$ will not match. This means that the test will be indeterminate. By similar reasoning, it can be shown that the test is indeterminate in the ordinary column difference rule as well. Thus, the column processing OR has two indeterminate rules. This triggers an impasse.

In effect, the solver can't decide what to do because neither rule is clearly true or clearly false. The crossed-out T<B column doesn't look exactly like a crossed-out T>B column or a non-crossed-out T<B column. Since the interpreter can't decide, it has the Force repair make the decision.

The Force repair is very simple: It chooses a rule that hasn't been executed and pushes the rule's action onto the stack. This causes the rule's action to be executed when interpretation resumes. In this case, the rules that Force must choose between have indeterminate applicability conditions. Force chooses the ordinary column-processing rule. This ultimately generates the bug shown above. Several other bugs have similar derivations, with different choices for the repair. The existence of these bugs supports the use of version spaces as applicability conditions.

Appendix 8
The Interstate Reference Problem

There seem to be good reasons for representing focus of attention explicitly (see section 11.1). Yet it would be nice, in one sense, if this were not the case. Representing internally something that has external meaning leads into a real rats' nest of problems involving reference across state changes. This problem will be discussed in the context of mathematical skills, of course, but the problem is more general. I have no good solution for it, but it is important enough that it deserves discussion. To put it concretely, the choice concerns what the representation should be for instances of notational objects. That is, what is it that fetch patterns retrieve? What is it that flows through the procedure's data flow pathways? How does one represent focus of attention when what is being attended to is a part of the problem state image? Most importantly, what happens to these internally stored objects when the external things that they refer to change?

States and state changes

Foci of attention refer into changeable problem states. Before asking what a focus of attention is, we need to ask what problem states and problem state changes are. In Newell and Simon's approach (1972), a theorist-supplied *problem space* specifies the representation for problem states and state change operators. Problem spaces can be different for different tasks and even for different subjects performing the same task. There is a problem with this approach that is best described by an example (for other problems, see section 13.1). The LEX (Mitchell et al., 1981; 1983) and ALEX (Neves, 1981) learners use a problem space representation of integral and algebraic equations. They represent a problem state as a tree. A tree representing the problem state

```
2x = 9-5
```

is shown in figure A8-1a. The state change operators in these two learners are tree pruning operations. When the subject adds a new line to the problem state above, yielding

```
2x = 9-5
2x = 4
```

the algebraic transformation underlying the subject's writing actions is represented by replacing a certain node in the problem state tree. Figure A8-1b shows the tree that represents the new problem state. The expression node of *a* has been replaced by a constant node, 4. All the state change operators in these two learners are similar. They modify a tree that represents the last equation on the worksheet.

What a human subject actually does, of course, is write successive lines, copying the unchanged portion of the equation from one line to the next. Yet, for LEX and ALEX, such state changes are not copying, but mutation. Essentially, their representation embeds, in a tacit way, the students' belief that the current focus of attention is the lowest equation in a vertical list. That is, a piece of student knowledge is represented *outside* the grammar and the procedure. It is hidden in the data structure used to represent the problem space (i.e., a tree) and the primitive operators used to represent writing actions. Some way of keeping student knowledge out of problem states and state change operators is needed. This is easily done.

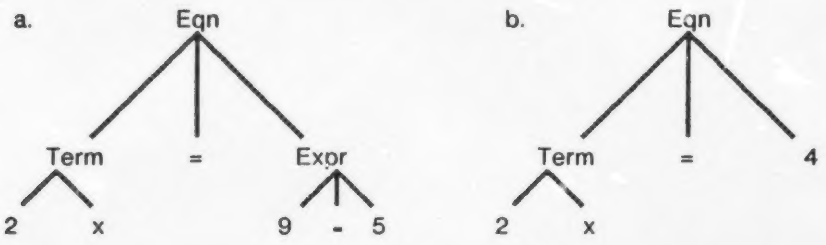


Figure A8–1
Trees representing problem states before and after an algebraic transformation.

Since the tasks covered by this theory are written symbol manipulation problems of various kinds, the grain size can be set at the level of individual symbols (i.e., letters, digits, arithmetic signs, brackets, horizontal bars, etc.). A problem state is represented as a set of symbol-location pairs, where a symbol's location is represented, say, by the Cartesian coordinates of the symbol's lower left corner and its upper right corner. The state change operations all represent writing specified symbols at specified locations. Writing a symbol is represented by adding an element to the set that represents the problem state. This grain size is small enough to be universal across subjects and tasks, and yet large enough to avoid character recognition and other theoretically irrelevant activities. The following hypothesis formalizes the particular approach adopted by the theory:

> *Universal state change operators*
> The set of state change operators (primitive goals) is universal across tasks and subjects. In particular, the state change operators are
>
> | | |
> |---|---|
> | `(Write x s)` | Writes symbol s in location `x`. |
> | `(CrossOut x)` | Draws a scratch mark over the object `x`. |
> | `(Bar x)` | Draws a horizontal or vertical line between the boundaries of the parts of the aggregate object `x`. |

There is a reason why there are three primitive operators instead of just `Write`. As discussed in section 15.2, the syntax of scratch marks and bars is somewhat different than other symbols. In particular, scratch marks are the only mathematical symbols that are placed *on top* of other symbols. All the other symbols do not overlap each other. Bars also have unique properties. Vertical and horizontal bars are often shared, in a certain sense, by several aggregate objects. For instance, the bar in subtraction problems is shared by all the columns in the problems. One conjecture, due to Jim Greeno (personal communication), is that bars are not symbols per se, but instead are used to mark the boundaries between aggregate objects.

One of the entailments of this small grain size is that procedures must express the students' actions in more detail. Instead of calling a large-grained state change operator, the procedure now calls a subprocedure that performs several writing actions. For instance, to represent decrementing a number in subtraction, the procedure might call a subprocedure that (1) crosses out a digit, and (2) writes the new digit vertically above the old one. In algebra equation solving, instead of calling a tree pruning operation that represents the state change from *a* to *b*,

```
a.    2x = 9-5                    b.    2x  =  9-5
                                        2x  =4
```

the procedure calls (1) a subprocedure that copies the left side of the equation to the next line down, (2) a writing action that writes the equals sign on the new line, and (3) a writing action that writes the 4 on the new equation's right side. When the grain size is fixed at the symbol level, the procedures must express knowledge about writing notation in more detail.

The frame problem

So far, all that's been done is to plug a leak in the theory's explanatory adequacy by removing a parameter from the theorist's control and making it a fixed part of the model instead. However, there are entailments of this move. They involve the notorious frame problem of AI. To illustrate how the frame problem arises in this domain, we'll return to the algebra equation solve as it makes the state change from state *a* to state *b*:

```
a.        2x = 9-5                    b.    2x  =  9-5
                                            2x  =  4
```

Suppose that the solver pushes an instance of a goal, solve-equation, at state *a*, then it goes on to find out how to solve the equation. Suppose further that goal has the equation of state *a* as its argument. At the time the solver comes to finally make the state change that yields state *b*, the goal instance and its argument are deep in the stack. The goal's argument is some stored focus of attention. What does it designate now? More importantly, what should it designate when the stack pops back to the goal instance, and the solver checks to see if the goal has been satisfied, i.e., whether the equation has been solved? There are several possible choices:

1. The symbols "2x = 9 − 5" in state *a*.
2. The symbols "2x = 9 − 5" in state *b*.
3. The symbols "2x = 4" in state *b*.

The first choice has data flow objects (foci of attention) referring to symbols in past problem states. For computer problem solvers, this is quite possible to implement, but it is wild as a conjecture about what people do! Choice 2 reflects the "fact" that objects such as written symbols somehow mangage to persist over time without changing their identity, possibly because the symbols have not moved. (But what if the student jiggles the paper? There are deep problems here.) A focus that was forged in state *a* refers in state *b*. However, algebra procedures are most simply expressed when choice 3 is used. This choice is exactly what LEX and ALEX use. When control pops back to the saved goal instance, its argument now refers to the latest version of the equation. This makes it easy to check whether the solve-equation goal is now satisfied. However, as just shown, this convention embeds knowledge (i.e., that the rule that the lowest equation in a vertical list is the one that counts as the current version of the equation) which ought to be expressed explicitly in the student's knowledge state. So choice 3 is not really tenable, and choice 1 is crazy. That leaves choice 2, which is (roughly) the one that Sierra uses. So the essence of the frame problem in this domain is representing interstate reference: What should happen to parts of the internal state that refer (somehow) to the external state, when that external state changes out from under them? The rest of the appendix discusses two different ways to achieve interstate reference.

Intensions vs. extensions

All notational objects share the characteristic that when they are instantiated in the problem state, they fill a certain area in the problem state. Hence, one way to represent focus of attention is as the actual region of the problem state, a rectangle in Cartesian coordinates, say, that the object fills. (This is a particularly appealing solution for mathematical forms because almost all of them have the same shape: rectangular. Forms for faces and stick figures aren't so computationally tractable.) Using a representation that stands for actual space on the problem state amounts to an *extensional* treatment of focus. When a description is matched during the execution of the procedure, its extension (referent) is found; from then on, the procedure passes the extension around.

The obvious alternative is to represent focus *intensionally.* Like a Montague grammar's interpretation of a natural language sentence, executing a description yields an intension, which is henceforth passed through the data flow paths. Only when a referent is really needed is the intension extended. In the case of mathematical procedures, there are two places where referents are needed: (1) The primitive reading operation, `Read`, needs an extension so that it can read a location in the problem state and return the symbol, if any, that occupies the location. (2) Primitive writing operations, e.g., `Write`, need an extended form so that they can actually write in the location specified by the form. Reading and writing are, of course, quite common in mathematical procedures. Quite likely, every description's intension is eventually extended either as a part of a larger intension or alone. (In this respect, mathematical procedures are probably quite different from natural language.) Hence, another way to view the intensional representation of focus is as a *lazy evaluation* scheme (Henderson & Morris, 1976). "Matching" a description (pattern) just wraps it around the values of the goal arguments that are used in the pattern (i.e., the input variables of the pattern). An intension looks just like a very deeply nested description with no arguments; only when the intension is actually used is this description extended. In short, the choice between extensions and intensions comes down to *when* descriptions are extended: at execution time or at read/write time.

As the remainder of this appendix will show, there are arguments against both sides. Roughly speaking, if descriptions are extended at execution time, the problem state can change before the extension is used so it may no longer be an accurate reflection of the meaning of the description. On the other hand, if descriptions are not fully extended until their time of use, the problem state may have changed in such a way that the intension mis-extends. In both cases, changes in the problem state between execution and use are fouling up the description-referent connection. With some effort, it can be shown that this difficulty has empirical ramifications for the theory. Hence, it is not a moot point, one that the theory can ignore.

The extensional approach

In chapter 11, it was shown that focus can be saved on the goal stack as the argument of a goal. If focus is represented extensionally, changes to the problem state can make some stored foci become out of date. Take addition problems, for example. Suppose a problem that starts with two columns, as *a* does,

```
a.     5 1          b.     1 5 1      c.     1 5 1
     + 9 6               +   9 6           +   9 6
     -----               -------           -------
                           1 4 7               4 7
```

grows to three columns, as in *b*. This growth will make the rectangle circumscribing the column sequence too small at the time termination is tested. Taken literally, this would force the procedure to believe the tens column was the last one, causing it to exit too soon, giving the answer shown in *c*. Moreover, it would be impossible to represent a procedure that gives the correct answer except by always including one more column in the original extension of the column-sequence form than the columns that are actually visible. This sort of look-ahead accommodation for an exception situation is difficult, if not impossible, to learn within the local learning framework erected by the theory. In short, the theory wouldn't be able to learn *any* correct procedure for addition. This false prediction is a rather damning entailment of adopting the extensional approach.

The intensional approach

The intensional-focus hypothesis has similar problems with stored foci going awry. Changes in the state of the problem state can cause the intension for a description that would extend correctly at matching time to extend incorrectly at the time of its use. An example will illustrate this problem. Suppose that a subtraction procedure describes the place where an answer should be written as the "rightmost column that I haven't written in yet." For subtraction without borrowing, this is an adequate description. Indeed, it is an excellent one when the procedure is equipped to suppress leading zeros. In problems such as *a*,

```
                                                6 14
a.   3 4 5 1 7 2        b.     3 7 4      c.     3 7 4
   - 1 4 5 1 7 1             - 1 2 8           - 1 2 8
   -------------             -------           -------
               1                                     6
```

the procedure must traverse quite a few columns away from the tens column's zero before it can determine that it should actually write the zero instead of suppress it. In problem *b*, the intension "rightmost column that I haven't written in" would refer to the units column at the time it's created, namely, at the beginning of the units column's processing. However, by the time borrowing is done and the intension is used to write the units column's answer, the intension now refers to the hundreds column since both the units and tens have been marked up. This would yield the star bug shown in *c* (given some further assumptions, which I won't describe here, that cause it to quit early). Once again, the content of a stored focus has "gone bad" due to changes in the state of the problem state between creation and use.

More star bugs are generated when intensions are repaired. One example involves the following conjunction of events: the same intension is used in two places; both uses cause impasses; and different refocus repairs are applied at each impasse. Roughly speaking, non-star bugs in such double-use cases require both refocusings to go the same direction. But such inter-impasse communication would violate the repair-impasse independence principle, and cannot be permitted. Extensional-foci avoid the problem because the repair unit can only apply the refocus/relaxation repair *after* it has popped the stack back to the description that originated the errant focus. Since intensional foci can be repaired *in situi*, there is no reason to back the stack up, yet repair without backing up leads to star bugs.

To sum up, there are problems for both the extensional focus and the intensional focus approaches. It seems that extending descriptions just once, either at execution time or at usage time, is insufficient. This urges considering a representation of focus that extends descriptions twice, both at creation and use. This is what the current version of Sierra does. It is not a true solution since it occasionally fails to work in ways that are clearly not ways that a human's procedure would fail. For completeness, the system will be described, but I make no claims for it's theoretical merit.

Using parse nodes to solve the interstate reference problem

The solution depends on the fact that the interface has a grammar separate from the patterns on the rules. This design is argued for in chapter 13. Because the grammar makes it possible to give a global parse to the problem state, focus can be represented by a node in the parse. A parse node has a box in Cartesian coordinates associated with it, so it is somewhat like an extension. However, it also has connections via the links in the parse tree to other nodes. In particular, it has links to its constituents. Using parse nodes as foci takes care of most of the difficulties mentioned above.

Since no trace of the description remains in the focus, when impasses that are caused by misplaced foci are repaired via the refocus repair, the local problem solver must first back the stack up until it gets to the description. In this respect, parse node foci mimic extensional foci and avoid one pit that intensional foci fall into. Parse nodes also escape the other way that intensions lead to star bugs. The details will be surpressed here.

The problem with the extensional focus approach was that the size of an aggregate object could grow while a box was being held on the stack, making it too small at the time of its use. If using a parse node meant simply using the box associated with it, the parse node approach would be subject to the same difficulties with growing images that the extensional foci approach had. What if it didn't use the box, but instead used the links from the node to the other parse nodes? In particular, suppose that "use" of a parse node involved matching it against a current parse of the problem state to find its closest approximation. Thus, if the problem state had not changed in any significant way, an exact isomorph of the node would be found. If it had, there would be a good chance that matching would find the "intended" node, the one that its description would reference if the description were extended at usage time (roughly speaking).

What this leads up to is an interpretation process that obeys the applicative hypothesis in a very literal way. It does not maintain a parse of the problem state in some global resource. Instead, the problem state is continually being parsed for various reasons. Perhaps a little story would help make this seem plausible. Imagine that people are equipped with a visual processor that is so powerful that glancing at an algebraic expression is sufficient to parse it. Whenever a description needs to be matched, a glance causes a parse tree to spring up, the description is matched against it, and a pointer to the best fitting node is returned to begin its route through the data flow pathways of the procedure. The portion of the parse tree that is not being pointed at gradually fades away. (One should keep firmly in mind that this is fiction.) Later, when the parse node has to be used, it's obsolete; the box it mentions has coordinates relative to the person's head position that are now quite different; connection to the visual world has been lost over time. So the reading or writing operation sends the parse node out to the visual processor, which glances at the problem state, gets a new parse tree, matches the node against the tree, and returns the best fitting node with a bona fide current (up to the millisecond!) box. This box the read/write operator can and does use.

The point of this story is to give some plausibility to the notion that adherence to the applicative hypothesis should be maintained by frequent parsing despite the fact that any computer scientist would blanch at such an extreme position on the space/time tradeoff.

Appendix 9
The Backup repair and the goal stack

This appendix contains several arguments concerning the goal stack. The arguments have been banished from their proper places in chapters 9 and 11 because they are rather long. They have been put together in this appendix because they all use the Backup repair. The appendix begins by showing that the Backup repair exists. Then, section 2 uses Backup to show that a finite state control regime will not suffice. Control flow must be recursive. The interpreter must use a goal stack. Section 3 uses Backup to show that data flow must be locally bound. It must be kept on the goal stack along with the instantiations of goals. Global binding, i.e., passing data in registers, will not allow Backup to function in the way that it has been observed to function.

A9.1 The Backup repair exists

Control and data flow are not easily deduced by observing sequences of writing actions. Too much internal computation can go on invisibly between observed actions for one to draw strong inferences about control or data flow. What is needed is an event which can be assumed or proven, in some sense, to be the result of an elementary, indivisible operation. The instances of this event in the data would shed light on the basic structures of control and data flow. Such a tool is found in a particular repair called the Backup repair. It bears this name since the intuition behind it is the same as the one behind a famous strategy in problem solving: backing up to the last point where a decision was made in order to try one of the other alternatives. This repair is so crucial in the remaining arguments that it is worth a few pages to defend its existence.

The existence argument begins by demonstrating that a certain six bugs should all be generated from the same core procedure. There are two arguments for this lemma. From the lemma, it is argued that the Backup repair is the best way to generate the six bugs.

A Cartesian product of bugs

The bugs all lack an ability to borrow from zero. (Henceforth, "BFZ" will be used to abbreviate "borrow from zero.") For easy reference, the six bugs will be broken into two sets called big-BFZ and little-BFZ. Big-BFZ bugs seem to result from replacing the whole column processing subprocedure whenever the column requires borrowing from a zero. Its bugs are:

Smaller-From-Larger-Instead-of-Borrow-From-Zero:	$\begin{array}{r} 3\,4\,5 \\ -\,1\,0\,2 \\ \hline 2\,4\,3 \end{array}\ \checkmark$	$\begin{array}{r} 3\,\overset{3}{4}{}^{1}5 \\ -\,1\,2\,9 \\ \hline 2\,1\,6 \end{array}\ \checkmark$	$\begin{array}{r} \overset{2}{3}{}^{1}0\,7 \\ -\,1\,6\,9 \\ \hline 1\,4\,2 \end{array}\ \times$
Zero-Instead-of-Borrow-From-Zero:	$\begin{array}{r} 3\,4\,5 \\ -\,1\,0\,2 \\ \hline 2\,4\,3 \end{array}\ \checkmark$	$\begin{array}{r} 3\,\overset{3}{4}{}^{1}5 \\ -\,1\,2\,9 \\ \hline 2\,1\,6 \end{array}\ \checkmark$	$\begin{array}{r} \overset{2}{3}{}^{1}0\,7 \\ -\,1\,6\,9 \\ \hline 1\,4\,0 \end{array}\ \times$
Blank-Instead-of-Borrow-From-Zero:	$\begin{array}{r} 3\,4\,5 \\ -\,1\,0\,2 \\ \hline 2\,4\,3 \end{array}\ \checkmark$	$\begin{array}{r} 3\,\overset{3}{4}{}^{1}5 \\ -\,1\,2\,9 \\ \hline 2\,1\,6 \end{array}\ \checkmark$	$\begin{array}{r} \overset{2}{3}{}^{1}0\,7 \\ -\,1\,6\,9 \\ \hline 1\,4\ \ \end{array}\ \times$

(The small numbers stand for the student's scratch marks. Correctly answered problems are marked with √ and incorrectly answered problems are marked with ×.) When a column requires borrowing from zero, as the units column does in the last problem, the first bug takes the absolute difference instead of borrowing. The second bug answers the column with the maximum of zero and the difference, namely zero. The third bug just leaves such BFZ columns unanswered. Notice that all three bugs perform correctly when the borrow does not require BFZ, as in the tens column of the last problem.

The little-BFZ bugs have a smaller substitution target, speaking roughly. Only the operations that normally implement borrowing across the zero are replaced, namely the operations of changing the zero to nine and borrowing from the next digit to the left. The three little-BFZ bugs are:

Borrow-Add-Decrement-Instead-of-Zero:

$$\begin{array}{cccc} & 3 & 4 & 5 \\ - & 1 & 0 & 2 \\ \hline & 2 & 4 & 3 \end{array}\ \checkmark \qquad \begin{array}{cccc} & & 3 & \\ & 3 & 4 & {}^{1}5 \\ - & 1 & 2 & 9 \\ \hline & 2 & 1 & 6 \end{array}\ \checkmark \qquad \begin{array}{cccc} & 2 & 1 & \\ & 3 & 0 & {}^{1}7 \\ - & 1 & 6 & 9 \\ \hline & 1 & 5 & 8 \end{array}\ \times$$

Zero-Borrow-At-Zero:

$$\begin{array}{cccc} & 3 & 4 & 5 \\ - & 1 & 0 & 2 \\ \hline & 2 & 4 & 3 \end{array}\ \checkmark \qquad \begin{array}{cccc} & & 3 & \\ & 3 & 4 & {}^{1}5 \\ - & 1 & 2 & 9 \\ \hline & 2 & 1 & 6 \end{array}\ \checkmark \qquad \begin{array}{cccc} & 2 & 0 & \\ & 3 & 0 & {}^{1}7 \\ - & 1 & 6 & 9 \\ \hline & 1 & 4 & 8 \end{array}\ \times$$

Stops-Borrow-At-Zero:

$$\begin{array}{cccc} & 3 & 4 & 5 \\ - & 1 & 0 & 2 \\ \hline & 2 & 4 & 3 \end{array}\ \checkmark \qquad \begin{array}{cccc} & & 3 & \\ & 3 & 4 & {}^{1}5 \\ - & 1 & 2 & 9 \\ \hline & 2 & 1 & 6 \end{array}\ \checkmark \qquad \begin{array}{cccc} & 2 & & \\ & 3 & {}^{1}0 & {}^{1}7 \\ - & 1 & 6 & 9 \\ \hline & 1 & 4 & 8 \end{array}\ \times$$

In the first case, absolute difference has been substituted for borrow's decrement. Hence, the zero in the third problem is changed to the absolute difference of zero and one, namely one. The second bug, Zero-Borrow-At-Zero, is generated by substituting the max-of-zero-and-difference operation for borrow's decrement. This causes the bug to cross out the zero of the last problem, and write a zero over it. The third bug, Stops-Borrow-At-Zero, simply skips the borrow-from part of borrowing when it is a zero that is to be decremented. (Zero-Borrow-At-Zero generates exactly the same answer as Stops-Borrow-At-Zero. The scratch marks are the only way to tell them apart. Since DEBUGGY is not given access to the scratch marks, it does not distinguish between the two bugs. Both are called Stops-Borrow-At-Zero.) The following table summarizes the bugs in a provocative way:

	Big-BFZ	Little-BFZ
Absolute diff.	Smaller-From-Larger-Instead-of-Borrow-From-Zero	Borrow-Add-Decrement-Instead-of-Zero
Max of zero, diff.	Zero-Instead-of-Borrow-From-Zero	Zero-Borrow-At-Zero
Noop	Blank-Instead-of-Borrow-From-Zero	Stops-Borrow-At-Zero

The Cartesian product relationship between these six bugs is exactly the kind of pattern that local problem solving captures. The most straightforward way to formalize it would be to postulate two core procedures, one for big-BFZ and another for little-BFZ. However, all six bugs miss the same kind of problems, namely just those problems that require borrowing from a zero. Intuitively, they seem to have the same cause: The subskill of borrowing across zero is missing from the subject's knowledge. This intuition is supported by the fact that subtraction curricula generally contain lessons that teaches borrowing from non-zero digits, followed by separate lessons that teach BFZ. The incremental learning hypothesis (section 3.6) implies that all core procedures associated with a certain segment of the lesson sequence will miss just the problems that lie outside the set of examples and exercises of that segment. If two core procedures were used, then both core procedures would have to be paired with the instructional segment that teaches borrowing from non-zero digits. This means that whatever the mechanism is that implements learning, it must explain how it could generate *two* core procedures that differed *only by how much of the procedure was repaired.* This approach amounts to keeping the local problem solver simple by making the learner more complex. On the other hand, the learner can be kept simple by having it generate just one core procedure instead of two, then making the local problem solver a tad more complex by having it generate all six bugs from that single core procedure. In short, it seems that either the learner or the solver must be made complex. This argument doesn't say which one it should be. The next argument shows that it should be the solver that should be made complex.

Bug migration evidence for deriving all six bugs from the same core procedure

The next argument is based on an assumption about bug migration: If a student migrates between two bugs, it is assumed that the two bugs are derived from the same core procedure via different repairs to the same impasses. This assumption is defended in chapter 6. Given it, all that has to be done to show that the big-BFZ bugs come from the same core procedure as the little-BFZ bugs is to find a case of a big-BFZ bug migrating with a little-BFZ bug. Figure A9–1 exhibits such a migration.

Figure A9–1 shows the first six problems of a subtraction test taken by subject 19 of classroom 20. This third grader gets the first four problems right, which involve only simple borrowing. He misses the next two, which require borrowing from zero. Crucially, these two problems are solved as if the subject had two different bugs from the Cartesian product pattern. This is an instance of intra-test bug migration. The fifth problem is solved by a little-BFZ bug: The student hits the impasse (note the scratch mark through the zero), and repairs it by skipping the decrement, a repair that generates the bug Stops-Borrow-At-Zero. He finishes up the rest of the problem without borrowing — apparently he wants to "cut his losses" on that problem. On the next problem, he again hits the decrement zero impasse, but repairs it this time by backing up and taking the absolute difference in the column that originated the borrow, the units column. This generates the bug Smaller-From-Larger-Instead-of-Borrow-From-Zero. Since both bugs are in the same bug migration class, both are somehow derived from the same core procedure via repair.

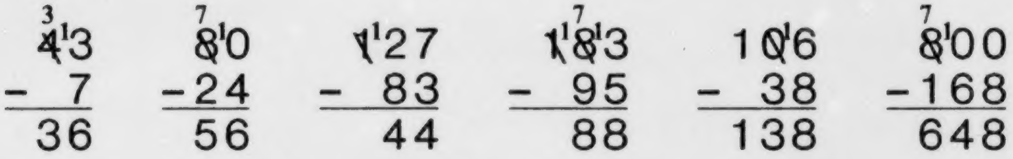


Figure A9–1
First six problems of a test that shows a bug migration.

Backup explains the Cartesian product pattern

Two arguments have forced the conclusion that all six bugs come from the same core procedure. Now the problem is to find repairs that will generate all of them, given that they all stem from the same impasse. One way to do that would be to use six separate repairs. However, that would not capture a fact about these bugs that was highlighted in their description: they fall into a Cartesian product pattern whose dimensions are the "size" of the patch (i.e., just the decrement, versus the whole borrowing operation) and its "function" (i.e., taking the absolute difference, versus skipping an operation, versus taking the maximum of zero and difference). It will be shown that this pattern can be captured by postulating a Backup repair, and hence that the Backup approach is more descriptively adequate.

The Backup repair resets the execution state of the interpreter back to a previous decision point in such a way that when interpretation continues, it will choose a different alternative than the one that led to the impasse that Backup repaired. The Backup repair is used for the big-BFZ bugs. Using Backup in those cases causes a secondary impasse. The secondary impasse is repaired with the same repairs that are used for the little-BFZ bugs. This is perhaps a little confusing, so it is worth a moment to step through a specific example.

Figure A9-2 is an idealized protocol of a subject who has the bug Smaller-From-Larger-Instead-of-Borrow-From-Zero. The (idealized) subject does not know about borrowing from zero. When he tackles the problem 305-167, he begins by comparing the two digits in the units column. Since 5 is less than 7, he makes a decision to borrow (episode *a* in the figure), a decision that he will later come back to. He begins to tackle the first of borrowing's two subgoals, namely borrowing-from (episode *b*). At this point, he gets stuck since the digit to be borrowed from is a zero and he knows that it is impossible to subtract a one from a zero. He's reached an impasse. The Backup repair gets past the decrement-zero impasse by "backing up," in the problem solving sense, to the last decision which has some alternatives open. The backing up occurs in episode *c*, where the subject says "So I'll go back to doing the units column." In the units column he hits a second impasse, saying "I still can't take 7 from 5," which he repairs ("so I'll take 5 from 7 instead"). He finishes up the rest of the problem without difficulty. His behavior is that of Smaller-From-Larger-Instead-of-Borrow-From-Zero. The other big-BFZ bugs would be generated if he had used different repairs in episode *c*. (e.g., Zero-Instead-of-Borrow-From-Zero would be generated if he reasoned, "I still can't take 7 from 5, but if I could, I certainly wouldn't have anything left, so I'll write 0 as the answer.")

Summary

It has been shown that the Backup repair is the best of several alternatives that generate the six bugs. It plays an equally crucial role in the generation of many other bugs, but the argument stuck with just six bugs for the sake of simplicity. Backup is the tool that will be used to support several hypotheses about core procedures.

a. $$\begin{array}{r} 305 \\ -\ 167 \\ \hline \end{array}$$

In the units column, I can't take 7 from 5, so I'll have to borrow.

b. $$\begin{array}{r} 305 \\ -\ 167 \\ \hline \end{array}$$

To borrow, I first have to decrement the next column's top digit. But I can't take 1 from 0!

c. $$\begin{array}{r} 305 \\ -\ 167 \\ \hline 2 \end{array}$$

So I'll go back to doing the units column. I still can't take 7 from 5, so I'll take 5 from 7 instead.

d. $$\begin{array}{r} \overset{2}{\cancel{3}}\overset{}{^{1}0}5 \\ -\ 167 \\ \hline 2 \end{array}$$

In the tens column, I can't take 6 from 0, so I'll have to borrow. I decrement 3 to 2 and add 10 to 0. That's no problem.

e. $$\begin{array}{r} \overset{2}{\cancel{3}}\overset{}{^{1}0}5 \\ -\ 167 \\ \hline 142 \end{array}$$

Six from 10 is 4. That finishes the tens. The hundreds is easy, there's no need to borrow, and 1 from 2 is 1.

Figure A9–2
Pseudo-protocol of a student performing the bug
Smaller-From-Larger-Instead-of-Borrow-From-Zero.

A9.2 Backup requires a goal stack

The Backup repair sends control back to some previous decision. The question is, which decision? There are three well known backup regimes used in AI:

1. *Chronological Backup:* The decision that is returned to is the one made most recently, regardless of what part of the procedure made the decision.
2. *Dependency-directed Backup:* A special data structure is used to record which actions depend on which other actions. When it is necessary to back up, the dependencies are traced to find an action that doesn't depend on any other action (an "assumption" in the jargon of Dependency-directed backtracking). That decision is the one returned to.
3. *Hierarchical Backup:* To support Hierarchical Backup, the procedure representation language must be hierarchical in that it supports the notion of goals with subgoals, and the interpreter must employ a goal stack. In order to find a decision to return to, Backup searches the goal stack starting from the current goal, popping up from goal to supergoal. The first (lowest) goal that can "try a different method" is the one returned to. Such a goal must have subgoals that function as alternative ways of achieving the goal, and moreover, some of these alternative methods/subgoals must not have been tried by the current invocation of the goal. When Backup finds such a goal on the stack, it resets the interpreter's stack in such a way that when the interpreter resumes, it will call one of the goal's untried subgoals. (In A.I., this is not usually thought of as a form of Backup. It is sometimes referred to by the Lisp primitives used to implement it, e.g., `THROW` in Maclisp, and `RETFROM` in Interlisp.)

The key difference among these backup regimes is, intuitively speaking, which decision points the interpreter "remembers." These establish which decision points the Backup repair can return to. In Chronological and Dependency-directed backtracking, the interpreter "remembers" all decision points. In Hierarchical backup, it forgets a decision point as soon as the corresponding goal is popped. The critical case to check is whether students ever back up to decision points whose corresponding goals would be popped if goal stacks were in use. If they don't return to such "popped" decision points, then Hierarchical Backup is the best model of their repair regime. On the other hand, if students do return to "popped" decisions, then either Chronological or Dependency-directed Backup is the better model. This section argues that students never back up up to popped decision points. By "never," I mean that returning to popped decision points generates star bugs. The evidence to be presented vindicates Hierarchical backup, and shows that (1) procedures' static structure has a goal-subgoal hierarchy, and (2) a goal stack is used by the interpreter in executing the procedure. In short, push down automata are better models of control structure than finite state automata.

Chronological Backup

Chronological Backup is able to generate the Backup bugs mentioned in the previous section. The walk-through of figure A9-2 should be evidence enough of that. However, by the impasse-repair independence principle, Chronological Backup can be used to repair any impasse. This causes problems. When Chronological Backup is applied to the impasses of certain independently motivated core procedures, it generates star bugs. The motivation for the core procedure in question is found in bugs such as:

Smaller-From-Larger-With-Borrow:

$$\begin{array}{rrrr} & & & \\ & 3 & 4 & 5 \\ - & 1 & 0 & 2 \\ \hline & 2 & 4 & 3 \end{array} \checkmark \qquad \begin{array}{rrrr} & & 3 & \\ & 3 & 4 & 5 \\ - & 1 & 2 & 9 \\ \hline & 2 & 1 & 4 \end{array} \times \qquad \begin{array}{rrrr} & 1 & 9 & \\ & 3 & 0 & 7 \\ - & 1 & 6 & 9 \\ \hline & 1 & 3 & 2 \end{array} \times$$

Zero-After-Borrow:

$$\begin{array}{rrrr} & & & \\ & 3 & 4 & 5 \\ - & 1 & 0 & 2 \\ \hline & 2 & 4 & 3 \end{array} \checkmark \qquad \begin{array}{rrrr} & & 3 & \\ & 3 & 4 & 5 \\ - & 1 & 2 & 9 \\ \hline & 2 & 1 & 0 \end{array} \times \qquad \begin{array}{rrrr} & 1 & 9 & \\ & 3 & 0 & 7 \\ - & 1 & 6 & 9 \\ \hline & 1 & 3 & 0 \end{array} \times$$

After completing the borrow-from half of borrowing, these bugs fail to add ten to the top of the column borrowed into (c.f., the top of the units column in the second problem). Instead, Smaller-From-Larger-With-Borrow answers the column with the absolute difference, and Zero-After-Borrow answers the column with zero. Apparently, and quite reasonably, the core procedure is hitting an impasse when it returns to the original column after borrowing. Since the column has not had ten added to the top digit as it should, the bottom digit is still larger than the top. This causes the impasse, which is seen being repaired two different ways in the two bugs.

With the core procedure thus independently motivated, one should find that a bug is generated when Chronological Backup is applied to the impasse. Instead, one finds a star bug. Consider a BFZ problem such as the last one above. The following is the first few problem states as the star bug solves it:

$$\text{a.}\ \begin{array}{rrrr} & 3 & {}^{1}0 & 7 \\ - & 1 & 6 & 9 \\ \hline \end{array} \qquad \text{b.}\ \begin{array}{rrrr} & 2 & & \\ & 3 & {}^{1}0 & 7 \\ - & 1 & 6 & 9 \\ \hline \end{array} \qquad \text{c.}\ \begin{array}{rrrr} & 2 & 9 & \\ & 3 & {}^{1}0 & 7 \\ - & 1 & 6 & 9 \\ \hline \end{array} \qquad \text{d.}\ \begin{array}{rrrr} & {}^{1}2 & 9 & \\ & 3 & {}^{1}0 & 7 \\ - & 1 & 6 & 9 \\ \hline \end{array} \qquad \text{e.}\ \begin{array}{rrrr} & & 8 & \\ & {}^{1}2 & 9 & \\ & 3 & {}^{1}0 & 7 \\ - & 1 & 6 & 9 \\ \hline \end{array}$$

First the core procedure borrows across zero (problem states *a*, *b*, and *c*), then it reaches the impasse in the originating column just prior to problem state *d*. It is going to back up to the most recent decision. Its most recent decision was that the digit that it was to borrow from in the hundreds column was non-zero and hence could be decremented, thereby finishing up the BFZ. This decision occurred just prior to *b*. Since it is the most recent decision chronologically, Chronological Backup causes control to go back to it and take its other alternative, which is to BFZ. The procedure starts the BFZ by adding ten to the hundreds column, as in state *d*. It tries to decrement the next column left, the thousands, but no such column appears. An impasse occurs. Suppose the student repairs it with the Noop repair, causing the decrement to be skipped. The BFZ continues, decrementing the tens (state *e*). When it gets done with this superfluous borrowing, it comes right back to the original column, which still has a too small top digit, so the impasse occurs again. If it is repaired with Chronological Backup again, control goes back to the hundreds again! Repeated uses of Chronological Backup results in an infinite loop, and a rather bizarre one at that. Even if the second occurrence is repaired some other way, going back to re-borrow in the first place is extremely odd behavior. Clearly, this is a star bug, and should not be predicted to occur by the theory. Chronological Backup is ruled out because it causes the theory to overgenerate.

Dependency-directed Backup

The basic idea of Dependency-directed Backup is to return to an action that doesn't depend on other actions. Dependency-directed Backup is really not a precise proposal for this domain until the meaning of "actions depending on other actions" is defined. Consideration of the star bug that was just described leads to a plausible definition. Part of what makes the star bug absurd is its going back to change columns to the left of the one where the impasse occurred. It seems clear that precondition violations in one column don't depend causally on writing actions in a different column. Suppose dependency between actions is defined to mean that the actions operate on the same column, or more generally, have the same locative arguments. Dependency-directed Backup will not make the mistake that Chronological Backup did. It will never go to another column to fix an impasse that occurs in this column.

However, even this rather vague definition limits Dependency-directed Backup too strongly. Several examples of Backup were presented earlier (i.e., the big-BFZ bugs) where the location was shifted. For instance, in the problem state sequence of figure 8-2, Smaller-From-Larger-Instead-of-Borrow-From-Zero shifted to a decision in the units column from a impasse in the tens column. Dependency-directed Backup can't generate such location-shifting Backups. Hence, it can't generate the big-BFZ bugs. As it is presently defined, it causes the theory to undergenerate.

Hierarchical Backup

In essence, these two approaches to backing up show that neither time nor space suffice. That is, such natural concepts as chronology or location will not support the kind of backing up that subjects apparently use. That leaves one to infer that Backup must be using some knowledge about the procedure itself. The issue that remains is what this extra knowledge is. A goal hierarchy is one sort of knowledge that will do the job, as the following demonstration shows.

The basic definition of Hierarchical Backup is that it can only resume decisions which are supergoals of the impasse that it is repairing. With this stipulation, any one of a number of goal structures will suffice to block the star bug that Chronological Backup generated as well as generate all the Backup bugs that have been presented so far. One such goal structure is shown in figure A9–3. This figure shows the goal-subgoal relationships with arrows. In this goal structure, the borrow-from goal (the goal that tests whether the digit to be borrowed from is zero) is not a supergoal of the Diff goal (the operation that reaches an impasse in the star bug's generation). There would be a chain of arrows from borrow-from to Diff if it were. Hence, when Backup occurs at the Diff impasse, it cannot go to the borrow-from decision even though that decision is chronologically the most recent.

So far, it has only been shown that the Backup repair obeys a hierarchy. It's not been shown that the goal hierarchy has anything to do with control of regular execution. But, if one simply postulated that the goal hierarchy existed off to the side, with no purpose other than to constrain Backup, then the theory is sorely damaged because there would be no way to test or refute this hypothesis. A more reasonable suggestion is that regular execution uses the hierarchical goal structure in such a way that Hierarchical Backup is a natural consequence of the structure of the internal, runtime information. The usual goal stack has this property. At any time, the goals on the stack are all supergoals of the currently executing goal. Postulating a goal stack naturally explains why Backup returns to supergoals — those are the only decision points that are explicitly saved in the runtime state. The goal stack architecture explains why the Backup repair behaves the way it does.

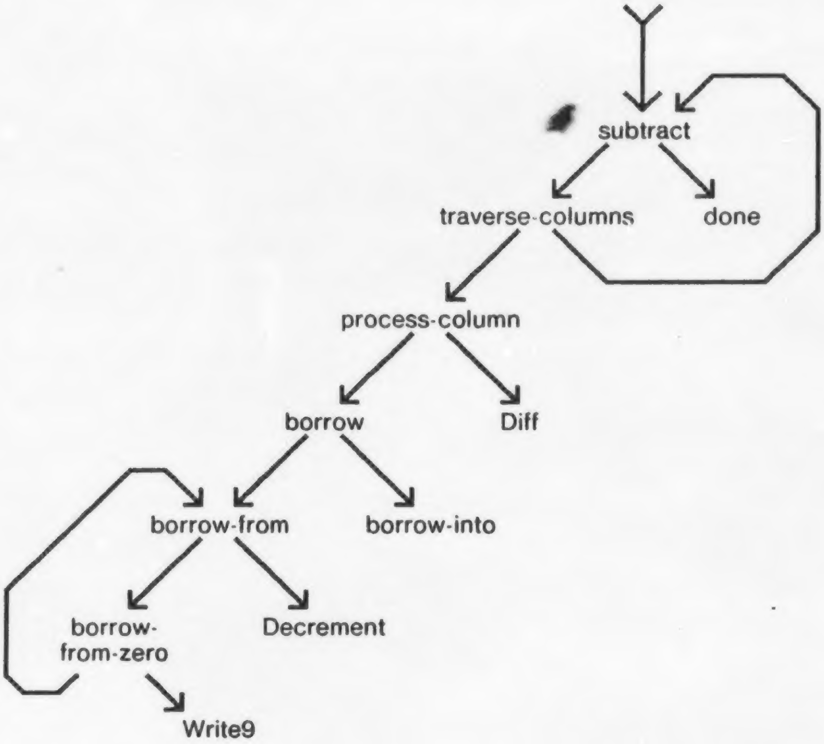

subtract
traverse-columns
done
process-column
borrow
Diff
borrow-from
borrow-into
borrow-from-zero
Decrement
Write9

Figure A9–3
A goal hierarchy that will allows Hierarchical Backup to function correctly.

Summary

First, the existence of a single core procedure for a set of six bugs was demonstrated. To generate the bugs, a Backup repair was postulated. Three kinds of Backup were considered. Chronological Backup used time to determine which decision points to back up to. Dependency-directed Backup used spatial locations. Presumably, these two natural kinds of information — time and space — would be accessible in a finite state machine architecture. However, they both proved ineffective in capturing the facts about the Backup repair's behavior. This showed that a new kind of information, a goal hierarchy, had to become a part of the procedures and a part of the execution state. That is, the finite state machine architecture had to be augmented with a goal stack. In short, to explain the Backup repair, the underlying control structure has to be that of a push down automaton.

- I. Fact1: Smaller-From-Larger-Instead-of-Borrow-From-Zero
  - A. The schema-instance architecture generates the bug
  - B. Registers
    - 1. A single register generates a star bug
    - 2. "Smart" Backup is irrefutable
    - 3. Multiple registers allows generation of the observed bug
- II. Fact2: Borrow-Across-Zero, left-ten
  - A. The schema-instance architecture generates the bug
  - B. Registers
    - 1. One register per goal: can't generate the bug
    - 2. One register per object: can't generate the bug
    - 3. "Smart" Backup is irrefutable
    - 4. Duplicate borrow-from goals: entails infinite procedure
    - 5. Duplicate borrow-from goals: equivalent to schema-instance

Figure A9–4

Outline of the argument between the register hypothesis and the schema-instance hypothesis

A9.3 Focus is locally bound

The argument in this section concerns how to represent focus of attention, or data flow as it was labelled in chapter 11. Two alternative architectures will be discussed: register-based and schema-instance. A register-based architecture is one that employs globally accessible memory resources, like the registers in a microprocessor. The schema-based architecture binds its memory resource into the control flow, just as a schema's instances hold extra information. This distinction will become clearer in a moment. As it turns out, just two facts are needed. One is the bug described earlier, Smaller-From-Larger-Instead-of-Borrow-From-Zero. The other fact will be introduced in a moment. The argument is organized around these two facts. Figure A9–4 is an outline of the argument.

It will be shown that the schema-instance architecture generates the first bug (I.A in the outline). However, the simplest version of the register-based architecture, the use of a single focus register, generates a star bug instead (I.B.1). Patching its difficulties by making the Backup repair more complex leads to problems with retaining the falsifiability of the theory (I.B.2). However, using several registers instead of just one register allows the bug to be generated simply (I.B.3). So the conclusion to be drawn from the first fact is that the register approach will be adequate only if there is more than one focus register.

The second part of the argument introduces a new bug involving the Backup repair. Once again, the schema-instance architecture predicts the facts correctly (II.A). Two different implementations of multiple registers fail (II.B.1 and II.B.2) by generating star bugs. A smarter Backup would fix the problem but remains methodologically undesirable (II.B.3). Postulating various complications to the goal structure of the procedure (II.B.4 and II.B.5) allow the correct predictions to be generated, but they have problems of their own. So the second part concludes that the register-based alternative is inadequate even when various complications are introduced. In overview, the argument is a nested argument-by-cases where all the cases except one are eliminated. To aid in following it, the cases will be labelled as they are in the outline of figure A9–4.

I.A Schema-instance generates the bug

The basic idea of a schema-instance architecture is that the location of a goal is strongly associated with the goal at the time it is first set. That is, when a goal such as borrowing is invoked, it is invoked at a certain column, or more generally, at a certain physical location in the visual display of the problem. In a schema-instance architecture, this association between goal and location, which is formed at invocation time, persists as long as the goal remains relevant. That is, the goal is a *schema*, which is *instantiated* by substituting specific locations, numbers or other data into it. If the control structure is known to be a stack, one would say that the focus is *locally bound.* Many computer languages, such as Lisp, have a schema-instance architecture: A function is instantiated by binding its arguments when it is called, and its arguments retain their bindings as long as the function is on the stack. However, in the interest of factoring the various parts of the theory independently, we will not assume that control is recursive.

The schema-instance architecture allows the bug of figure A9–1 to be generated quite naturally. Suppose the Sub1Col goal were strongly associated with its location, namely the units column, in some short term memory associated with Sub1Col's instantiation. Backup causes the resumption of the goal at the stored location. Another way to think of this is that the interpreter is maintaining a short term "history list" that temporarily stores the various invocations of goals with their locations. In regular execution, when the borrowing goal finishes, the Sub1Col goal is resumed *at the same place as it started.* That is, in the long-term representation of the procedure, the Sub1Col goal is a schema with its location abstracted out. It is bound to a location (instantiated) when it is invoked. It is the instantiated goal that Backup returns to, not the schematic one.

This schema-instance distinction, which is at the heart of almost all modern programming languages, entails the existence of some kind of temporary memory to store the instantiations of goals, and thus motivates this way of implementing the Backup repair. But there are of course other ways to account for focus shifting during Backup. Several will be examined and shown to have fewer advantages than the schema-instance one.

I.B.1 A single register architecture generates a star bug

Suppose that instead of using the schema instances to implement data flow, the architecture uses a single register, a you-are-here pointer to some place in the current problem state. There would be no problem representing the subtraction procedure in such an architecture: Actions in the procedure's representation would change the contents of this register as the various goals are invoked.

However, if the you-are-here register is simply left alone during backing up, then a star bug is generated. It is illustrated in figure A9–5. At episode (b), Backup resumes the Sub1Col goal, but the you-are-here register is not restored to the units column. Instead, the tens column is processed. The units column is left with no answer despite the fact that its top digit has had ten added to it. In the judgment of expert diagnosticians, this behavior would never be observed among subtraction students. It is a *star bug.* The theory should not predict its occurrence.

a. $\begin{array}{r} 4\cancel{0}2 \\ -\,106 \\ \hline \end{array}$ In the units column, I can't take 6 from 2, so I'll have to borrow. First I'll add ten to the 2.

b. $\begin{array}{r} 4\cancel{0}2 \\ -\,106 \\ \hline \end{array}$ I'm supposed to decrement the top zero, but I can't! So I guess I'll back up to processing the column.

c. $\begin{array}{r} 4\cancel{0}2 \\ -\,106 \\ \hline 0 \end{array}$ Processing it is easy: 0 – 0 is 0.

d. $\begin{array}{r} 4\cancel{0}2 \\ -\,106 \\ \hline 30 \end{array}$ The hundreds is also easy. I'm done!

Figure A9-5
Pseudo-protocol of a student performing a star bug

I.B.2 A smart Backup repair makes the theory too tailorable

To avoid the star bug, the Backup repair would have to employ an explicit action to restore the register to the units column in episode *c* of the protocol of figure A9–1. But how would it know to do this? Backup would have to determine that the focus of attention should be shifted rightward by doing an analysis of the goal structure contained in the stored knowledge about the procedure. It would see that in normal execution, a locative focus shifting function was executed between the Borrow-from goal and the Sub1Col goal. For some reason, it decides to execute the inverse of this shift as it transfers control between the two goals.

Not only does this implementation make unmotivated assumptions, but it grants Backup *the power to do static analyses of control structure.* This would give it significantly more power than the other repairs, which do simple, local things like skipping the stuck operation. Postulating a smart Backup gives the local problem solver so much power that one could "explain" virtually any behavior by cramming the explanation into the black boxes that are repairs. That is, it gives the theory too much tailorability. It is much better to make the repairs as simple as possible by embedding them in just the right architecture.

I.B.3 Multiple registers allow generation of the bug

Another way to implement Backup involves using a *set* of registers. The registers have some designated semantics, such as "most recently referenced column" or "most recently referenced digit." That is, the registers could be associated with the type or visual shape of the locations referenced, as Smalltalk's class variables are. Alternatively, they could be associated with the schematic goals. Old programming languages used to implement a subroutine's variables this way by allocating their storage in the compiled code, generally right before the subroutine's entry point. Sub1Col would have a register, Borrow would have a different register, and so on.

Given this architecture, Backup is quite simple. Returning to Sub1Col requires no locative focus shifting on its part. Since the Sub1Col register (or the column register, if that's the semantics) was not changed by the call to Borrow, it is still pointing at the units column when Backup causes control to return to Sub1Col. This multi-register implementation is competitive with the schema-instance one as far as its explanatory power. Backup is simple and local. Moreover, the data flow architecture has motivation independent of the Backup repair in that is used during normal interpretation. However, the multi-register approach fails to account for certain empirical facts that will now be exposed.

II.A Another bug, and schema-instance can generate it

The argument in this case, case II, is similar to that of case I. It takes advantage of subtraction's recursive borrowing to exhibit Backup occurring in a context where there are two instantiations of the Borrow goal active at the same time. There are two potential destinations for Backup. It will be shown that the schema-instance mechanism is necessary to make empirically correct predictions.

A common bug is one that forgets to change the zero when borrowing across zero. This leads to answers like:

$$\begin{array}{cccc} & {\scriptstyle 2} & & \\ & {\scriptstyle 3} & & \\ & 4^{1} & 0^{1} & 2 \\ - & 1 & 3 & 9 \\ \hline & 1 & 7 & 3 \end{array}$$

The 4 was decremented once due to the borrow originating in the units column, and then again due to a borrow originating from the tens column because the tens column was not changed during the first borrow, as it should have been. This bug is called Borrow-Across-Zero. It is a common bug. Of 417 students with bugs, 51 had this bug.

An important fact is seen in figure A9–6. The bug decrements the one to zero during the first borrow. Thus, when it comes to borrow a second time, it finds a zero where the one was, and performs a recursive invocation of the borrow goal. This causes an attempt to decrement in the thousands columns, which is blank. An impasse occurs. The answer shown in the figure is generated by assuming the impasse is repaired with Backup. This sends control back to the most recently invoked goal that has alternatives. When Backup is trying to decide which goal to return to, the active goals are

Borrow-from	(the recursive invocation located at the thousands column)
Borrow-from-zero	(at the hundreds column)
Borrow-from	(at the hundreds column)
Borrow	(at the tens column)
Sub1Col	(at the tens column)
Multi	
Sub	

In this core procedure, the Borrow-from-zero goal has no alternatives. It should always both write a nine over the zero and Borrow-from the next column, although here the write-nine step has been forgotten. The Borrow-from goal has alternatives because it has to chose between ordinary, non-zero borrowing and borrowing from zeros. Since Borrow-from was the most recently invoked goal that has alternatives left, Backup returns to it. Execution resumes by taking its other alternative, the one that was not taken the first time. Hence, an attempt is made to do an ordinary Borrow-from, namely a decrement. Crucially, this happens in the hundreds column, which has a zero in the top. The attempt to decrement zero causes a new impasse. We see that it is the hundreds column that was returned to because the impasse was repaired by substituting an increment for the blocked decrement, causing the zero in the hundreds column to be changed to a one.

The crucial fact is that Backup shifted the focus from the thousands column to the hundreds column, even though both the source and the destination of the backing up were Borrow-from goals. This shift is predicted by the schema-instance architecture. It takes only a moment to show that the empirical adequacy of the register architecture is not so high.

a.
$$\begin{array}{r} \overset{0}{\cancel{1}}0{}^{1}2 \\ -\;\;39 \\ \hline 3 \end{array}$$

Since I can't take 9 from 2, I'll borrow. The next column is 0, so I'll decrement the 1, then add 10 to the 2. Now I've got 12 take away 9, which is 3.

b.
$$\begin{array}{r} \overset{0}{\cancel{1}}0{}^{1}2 \\ -\;\;39 \\ \hline 3 \end{array}$$

Since I can't take 3 from 0, I'll borrow. The next digit is 0, but there isn't a digit after that!

c.
$$\begin{array}{r} \overset{0}{\cancel{1}}0{}^{1}2 \\ -\;\;39 \\ \hline 3 \end{array}$$

I guess I could quit, but I'll go back to see if I can fix things up. Maybe I made a mistake in skipping over that 0, so I'll go back there.

d.
$$\begin{array}{r} \overset{\overset{1}{\cancel{0}}}{\cancel{1}}0{}^{1}2 \\ -\;\;39 \\ \hline 3 \end{array}$$

When I go back there, I'm still stuck because I can't take 1 from 0. I'll just add instead.

e.
$$\begin{array}{r} \overset{\overset{1}{\cancel{0}}}{\cancel{1}}{}^{1}0{}^{1}2 \\ -\;\;39 \\ \hline 173 \end{array}$$

Now I'm okay. I'll finish the borrow by adding 10 to the ten's column, and 3 from 10 is 7. The hundreds is easy, I just bring down the 1. Done!

Figure A9–6
Pseudo-protocol of a student performing a variant of the bug Borrow-Across-Zero.

II.B.1 One register per goal can't generate the bug

Suppose each schematic goal has its own register. Borrow-from would have a register, and it would be set to the top digit of the thousands column at the first impasse (episode *b* in figure A9–6). Hence, if Backup returns to the first invocation of Borrow-from, the register will remain set at the thousands column. Hence, Backup doesn't generate the observed bug of figure A9–6. In fact, it can't generate it at all: The only register focused on the hundreds column is the one belonging to the Borrow-from-zero goal. That goal has no open alternatives, so Backup can't return to it. Even if it did, it wouldn't generate the bug of figure A9–6. So one register per goal is an architecture that is not observationally adequate.

II.B.2 One register per object doesn't generate the bug

Assuming the registers are associated with object types fails for similar reasons. Both impasses (episodes *b* and *d*) involve the same type of visual object, a digit, and hence the corresponding register would have to be reset explicitly by Backup in order to cause the observed focus shift.

II.B.3 Smart Backup makes the theory too tailorable

But providing Backup with an ability to explicitly reset registers would once again require it to do static analysis of control structure — an increase in power that should not be granted to repairs.

II.B.4 Duplicate borrow-from goals

One could object that we have made a tacit assumption that it is the same (schematic) Borrow-from that is called both times. If there were two schematic Borrow-froms, one for an adjacent borrow, and one for a borrow two columns away from the column originating the borrow, then they could have separate registers. This would allow Backup to be trivial once more. However, this argument entails either that one have a subtraction procedure of infinite size, or that there be some limit on the number of columns away from the originating column that the procedure can handle during borrowing. Both conclusions are implausible.

II.B.5 Duplicate borrow goals

One could object that there is another way to salvage the multiple register architecture. Suppose that the schematic procedure is extended by duplicating Borrow goals (plus registers) as needed. The bug could be generated, but this amounts either to a disguised version of schemata and instantiations, or an appeal to some powerful problem solver (which then has to be explained lest the theory lapse into infinite tailorability). So, this alternative is not really tenable either.

Conclusion

This rather lengthy argument concludes with the schema-instance architecture the only one left standing. What this means is that representations that do not employ the schemata and instances, such as finite state machines with registers or flow charts, can be dropped from consideration. This puts us, roughly speaking, on the familiar ground of "modern" representation languages for procedures, such as stack-based languages, certain varieties of production systems, certain message-passing languages, and so on.

Appendix 10
Satisfaction Conditions

This appendix presents arguments concerning what types the procedure representation language should allow for goals. It continues a line of argument begun in chapter 10. It contrasts two hypotheses:

1. And-Or: Goals have a binary type. If a goal's type is AND, all applicable subgoals are executed before the goal is popped. If it is OR, the goal pops as soon as one subgoal is executed.
2. Satisfaction conditions: Goals have a condition which is tested after each subgoal is executed. If the condition is true, the goal is popped. Metaphorically speaking, the goal keeps trying different subgoals until it is satisfied.

Satisfaction conditions were used in the version of repair theory presented in Brown and VanLehn (1980). First the arguments in their favor will be presented (some of these are repeated from chapter 10 so that the appendix will be relatively self-contained). Then arguments against them will be presented, and shown to be slightly stronger. The arguments concern how to constrain the deletion operator in such a way that it will continue to generate the deletion bugs (see chapter 7) but it will not generate certain star bugs. It has already been shown (in section 10.3) that limiting the operator to delete only AND rules allows it to generate all the deletion bugs while preventing it from generating many star bugs. However, it still allows a few star bugs to be generated. These star bugs will be examined in detail in order to motivate a way of blocking their deletion.

A10.1 Satisfaction conditions block certain star bugs

Suppose that the main loop of subtraction, which traverses columns, has the following goal structure when it is expressed using And-Or goal types. (The following is a translation of the goal structure of figure 10-1 into an And-Or representation. The rule numbers are the same.)

```
Goal: Multi (C) Type: AND
   3.  (Sub1Col C)
   4.  (SubRest (Next-column C)

Goal: SubRest (C) Type: OR
   5.  C is not the leftmost column ⇒  (Multi C)
   6.  C's bottom is blank ⇒  (Show C)
   7.  true ⇒ (Diff C)
```

The applicability conditions for the AND rules have been omitted. The applicability conditions for the OR goal, SubRest, reflect the fact that they are tested in order and only one is executed. For instance, if the column C is the leftmost column, then rule 5 will not be applicable, and control moves on to test rule 6. If that rule applies, the primitive Show answers the column, then control returns to SubRest. Since the goal is marked as an OR goal, and one rule has been executed, no more rules are tested. In particular, the default rule, 7, will not be tested. Instead, the SubRest goal is popped.

The Multi goal is an AND goal, so either of its subgoals can be deleted. Deleting rule 4 creates a bug that only does the units column. Intuitively, only doing one column would be the mark of a student who has not yet been taught how to do multiple columns. Since doing multiple columns is always taught before borrowing, it would be highly unlikely for a student to know all about borrowing and yet do only the units column. Hence, if all of BFZ (BFZ abbreviates "borrow from zero") were present when rule 4 is deleted, the procedure would generate a star bug. The work of this star bug appears below:

Only-Do-Units:

$$\begin{array}{rrr} & 3 & 2\ 9 \\ 3\ 4\ 5 & 3\ 4^{1}5 & 3^{1}0^{1}7 \\ \underline{-\ 1\ 0\ 2} & \underline{-\ 1\ 2\ 9} & \underline{-\ 1\ 6\ 9} \\ 3\ \times & 6\ \times & 8\ \times \end{array}$$

If borrowing were not yet learned and rule 4 were deleted, then reasonable bugs would be generated. For instance, one reasonable, but as yet unobserved bug does only the units column but it simply takes the absolute difference there instead of borrowing. This would be the bug set {Only-Do-Units Smaller-From-Larger}. In short, there is nothing wrong with the deletion of rule 4 *per se*, but it can create a procedure that mixes competence with incompetence in an unlikely manner.

Another star bug occurs when rule 13 is deleted from Borrow, given the following version of column processing and borrowing (this is also a translation of figure 10–1):

Goal: Sub1Col (C) Type: OR
8. T<B in C ⇒ (Borrow C)
9. the bottom of C is blank ⇒ (Show C)
10. true ⇒ (Diff C)

Goal: Borrow (C) Type: AND
11. (Borrow-from (Next-column C))
12. (Add10 C)
13. (Diff C)

Deleting rule 13 generates a procedure that sets up to take the column difference after a borrow, but forgets to actually take it. This leads to the following star bug:

*Blank-With-Borrow:

$$\begin{array}{rrr} & 3 & 2\ 9 \\ 3\ 4\ 5 & 3\ 4^{1}5 & 3\ 0^{1}7 \\ \underline{-\ 1\ 0\ 2} & \underline{-\ 1\ 2\ 9} & \underline{-\ 1\ 6\ 9} \\ 2\ 4\ 3\ \checkmark & 2\ 1\ \times & 1\ 3\ \times \end{array}$$

What makes this bug so unlikely is that it leaves a blank in the answer despite the fact that it shows a sophisticated knowledge of borrowing.

It is perhaps possible to put explicit constraints on conjunctive rule deletion in order to block the deletion that generate the star bugs. However, there is a second way to prevent overgeneration that will be shown to have some advantages. The basic idea is to make the operator inapplicable by changing the types of the two goals in question so that they are not AND goals. This would make the deletion operator inapplicable. That is, one changes the knowledge representation rather than the operator.

The proposed change is to adopt a new exit convention, the third one mentioned in the introduction. The exit convention generalizes the binary AND/OR type to become *satisfaction conditions.* The basic idea of an AND goal is to pop when *all* subgoals have been executed, while an OR goal pops when *one* subgoal has been executed. The idea of satisfaction conditions is to have a goal pop *when its satisfaction condition is true.* Subgoals of a goal are executed until either the goal's satisfaction condition becomes true, or all the applicable subgoals have been tried. (Note that this is not an iteration construct — an "until" loop — since a rule can only be executed once.) AND goals become goals with FALSE satisfaction conditions: Since subgoals are executed until the satisfaction condition becomes true (which it never does for the AND) or all the subgoals have been tried, giving a goal FALSE as its satisfaction condition means that it will always execute all its subgoals. Conversely, OR goals are given the satisfaction condition TRUE: The goal exits after just one subgoal is executed.

With this construction in the knowledge representation language, one is free to represent borrowing in the following way:

Goal: Sub1Col (C) Satisfaction condition: C's answer is non-blank:
8. T<B in C ⇒ (Borrow C)
9. the bottom of C is blank ⇒ (Show C)
10. true ⇒ (Diff C)

Goal: Borrow (C) Satisfaction condition: false
11. (Borrow-from (Next-column C))
12. (Add10 C)

The AND goal, Borrow, now consists of two subgoals. After they are both executed, control returns to Sub1Col. Because Sub1Col's satisfaction condition is not yet true — the column's answer is still blank — another subgoal is tried. Diff is chosen and executed, which fills in the column answer. Now the satisfaction condition is true, so the goal pops.

Given this encoding of borrowing, the conjunctive rule deletion operator does exactly the right thing when applied to borrow. In particular, since rule 13 is no longer present, it is no longer possible to generate the star bug, *Blank-with-Borrow-From by deleting it. Rule 13 has been merged, so to speak, with rule 10. Since rule 10 is under a non-AND goal, Sub1Col, it is protected from deletion.

Similarly, the star bug associated with column traversal can be avoided by restructuring the loop across columns. The two goals Multi and SubRest are replaced by a single goal:

Goal: SubAll (C) Satisfaction condition: C is the leftmost column.
5. true ⇒ (Sub1Col C)
6. true ⇒ (SubAll (Next-column C))

This goal first processes the given column by calling the main column processing goal, Sub1Col. Then it checks the satisfaction condition. If the column is the problem's leftmost column, the goal pops. Otherwise, it calls itself recursively. By using a satisfaction condition formulation, generation of the star bug is avoided. The AND goal, Multi, has eliminated along with its rule 3, the rule whose deletion caused the star bug.

These two illustrations indicate that augmenting the representation with satisfaction condition creates an empirically adequate treatment of deletion. Satisfaction condition were used for several years in Sierra (Brown & VanLehn, 1980; VanLehn, 1983). They also play a crucial role in the formulation of empirically adequate critics. A critic is a kind of impasse condition. Critics serve both to trigger repairs and to filter repairs. It turns out that one of the problems with critics can be solved using satisfaction conditions. In considering this problem, a different approach was discovered to deletion and deletion blocking that led to the position currently taken by the theory.

A11.1 The blank answer critic problem

One of the unsolved problems mentioned in Brown and VanLehn (1980) is called the *blank answer critic problem.* One function of critics is to filter out repairs that are applicable to the given impasse whenever they act in such a way as to immediately violate a critic. There is evidence that there is a critic, called the blank answer critic, that objects to answers that have blanks in them (i.e., a number that looks like "34 5" as an answer). The evidence for this critic involves a certain rule deletion that causes local problem solving.

In the reformulated version of Borrow, there are two rules. One does borrowing-into, the other does borrowing-from. If the rule that does borrowing-into is deleted, then the Add10 operation of borrowing is skipped. This means that the column difference operation that follows it will be confronted with a column that is still in its original T<B form. This causes an impasse. Two different repairs cause the following two bugs to be generated:

Smaller-From-Larger-With-Borrow:

$$\begin{array}{r} 3\,4\,5 \\ \underline{-\,1\,0\,2} \\ 1\,4\,3 \end{array} \checkmark \qquad \begin{array}{r} 3\,\overset{3}{4}\,5 \\ \underline{-\,1\,2\,9} \\ 2\,1\,4 \end{array} \times \qquad \begin{array}{r} \overset{2}{3}{}^{1}\overset{9}{0}\,7 \\ \underline{-\,1\,6\,9} \\ 1\,3\,2 \end{array} \times$$

Zero-After-Borrow:

$$\begin{array}{r} 3\,4\,5 \\ \underline{-\,1\,0\,2} \\ 1\,4\,3 \end{array} \checkmark \qquad \begin{array}{r} 3\,\overset{3}{4}\,5 \\ \underline{-\,1\,2\,9} \\ 2\,1\,0 \end{array} \times \qquad \begin{array}{r} \overset{2}{3}{}^{1}\overset{9}{0}\,7 \\ \underline{-\,1\,6\,9} \\ 1\,3\,0 \end{array} \times$$

The existence of these two bugs establishes the impasse occurs. However, if this impasse is repaired with the Noop repair, a repair that causes the stuck operation to be skipped, then the following star bug is generated:

*Blank-With-Borrow:

$$\begin{array}{r} 3\,4\,5 \\ \underline{-\,1\,0\,2} \\ 1\,4\,3 \end{array} \checkmark \qquad \begin{array}{r} 3\,\overset{3}{4}\,5 \\ \underline{-\,1\,2\,9} \\ 2\,1\, \end{array} \times \qquad \begin{array}{r} \overset{2}{3}{}^{1}\overset{9}{0}\,7 \\ \underline{-\,1\,6\,9} \\ 1\,3\, \end{array} \times$$

After a borrow-from, control returns to the column which initiated the borrow. That column is still in its original state. When the Diff operation comes to perform the column difference, it hits an impasse because it can't take a larger number from a smaller one. The Noop repair causes the stuck operation, in this case Diff, to simply be skipped. Hence, the column is left unanswered, creating the pattern of answers shown above. This pattern makes the bug a star bug, since such gapped answers are totally unlikely given the sophistication in borrowing.

The blank answer critic is supposed to block this bug by sensing that the Noop repair will leave a gap in the answer. The problem is that all the other known critics in subtraction are preconditions to primitive actions. That is, they are tested just before an action. They guard against errors of *commission.* However, the blank answer critic cannot be implemented as a precondition. The basic problem is that is must sense an error of *omission* rather than commission. At the time the Noop repair is being considered, the offending gap is on the left end of the answer (i.e., the answer string looks like " 34" with the blank on the left). It does not look like a gap yet. If the blank answer critic is a precondition, it must be very smart. It must be able to read the control structure of the procedure in order to tell that what is now a harmless boundary between digits and blanks will become a gap. In short, blocking the Noop repair with a blank answer critic forces the model to use a very powerful model of critics. If any other way can be found to block the star bug, this degree of freedom need not be added to the theory.

There is an aspect of gaps in the answers which reveals a way to solve the blank answer critic problem. It turns out that there is a bug that does leave blanks in the answer:

Blank-Instead-of-Borrow:

$$\begin{array}{r} 3\,4\,6 \\ \underline{-\,1\,0\,2} \\ 1\,4\,3 \end{array}\ \checkmark \qquad \begin{array}{r} 3\,4\,6 \\ \underline{-\,1\,2\,9} \\ 2\,2 \end{array}\ \times \qquad \begin{array}{r} 2\,0\,7 \\ \underline{-\,1\,6\,9} \\ 1 \end{array}\ \times$$

This bug is generated by assuming that borrowing hasn't been learned yet. When a column is processed that normally would require borrowing, an impasse occurs since a larger number can't be taken from a smaller one. Repairing with Noop leaves the answers to such columns blank. The occurrence of this bug juxtaposed with the non-occurrence of *Blank-With-Borrow shows that whatever is preventing gaps in the answers must have been *acquired.* In particular, it must have been acquired sometime between the stages of learning represented by the two core procedures. It must have been acquired sometime after borrowing was learned since Blank-Instead-of-Borrow hasn't learned about borrowing and it also hasn't learned about blanks in the answer.

Satisfaction conditions provide the hook that solves the mystery. Instead of a critic that is sensitive to blanks, one postulates that a repair is filtered out if it *causes a goal to exit unsatisfied.* More accurately, one postulates that there is an impasse condition that is true if (1) a goal pops due to the fact that all the applicable subgoals have been tried, and (2) that goal has a non-trivial satisfaction condition that is false. That is, the goal exits unsatisfied. The intuition behind this impasse condition is that if a goal has specific information that indicates when it is satisfied, and an attempt is made to exit it without these conditions being true, then it is obvious that something is wrong, and this triggers local problem solving.

In this case, gaps in the answer are prevented because the main column processing goal, Sub1Col, has a satisfaction condition that tests whether its answer is blank. It is satisfied only if the answer is blank. When the local problem solver is considering which repair to chose, it discovers that the Noop repair will leave a blank in the answer causing an attempt to exit Sub1Col with the answer left blank. Since the satisfaction condition is false, the new impasse condition is true. This means the Noop repair must be filtered since repairs are chosen only if they get the interpreter unstuck (i.e., no impasse condition is true).

The explanation of the acquisition phenomenon rests on assuming that the student who knows about borrowing has a non-trivial satisfaction condition while the student who has not yet learned borrowing has a trivial one. That is, the student who has not learned borrowing has a simple OR-type satisfaction condition for the goal that processes a column:

```
Goal: Sub1Col (C)  Satisfaction condition: true
  1.  C's bottom is blank  ⇒ (Show C)
  2.  true       ⇒ (Diff C)
```

whereas the student who has learned borrowing has the non-trivial satisfaction condition version of Sub1Col:

```
Goal: Sub1Col (C)  Satisfaction condition: C's answer is non-blank
  1.  T<B        ⇒ (Borrow C)
  2.  C's bottom is blank  ⇒ (Show C)
  3.  true       ⇒ (Diff C)

Goal: Borrow (C)  Satisfaction condition: false
  1.  (Add10 C)
  2.  (Borrow-from (Next-column C))
```

In the first case, where borrowing has not been taught and the second occurrence of Diff as following borrowing has not been seen, there isn't enough evidence to separate the simple OR type from a non-trivial satisfaction condition. Hence, the learner conservatively takes the simpler satisfaction condition. In the second case, borrowing has revealed a second occurrence of Diff. The student infers that the common goal of all the different ways to process a column is to make the answer be non-blank. This motivates the inclusion of the non-trivial satisfaction condition. This acquisitional story accounts for the fact that pre-borrow students allow blanks in their answers because they don't have the non-trivial satisfaction condition yet, and the post-borrow students block repairs that leave blanks because they do have the non-trivial satisfaction condition.

To summarize, once satisfaction conditions are permitted, they can be used two ways: (1) to block application of a deletion operator, and (2) to block the repairs that would leave satisfaction conditions unsatisfied. Moreover, one can account for the acquisition of the blank answer critic by the somewhat more natural acquisition of the satisfaction condition. There are a number of good things about this account, but there are also some severe flaws.

A10.2 Problems with satisfaction conditions

The first flaw lies in the "conservative" acquisition of satisfaction conditions. When satisfaction condition acquisition is spelled out in a little more detail, it is found to make wrong predictions. The basic idea is that a satisfaction condition is acquired when one learns a setup or preparation step for some main step. That is, if M is a known action and one learns that a certain sequence <X Y Z M> is an alternative to M, then one infers that M is a main step and that the first part of the sequence, <X Y Z>, is preparation for the main step M. For instance, the main step of column processing is Diff and the preparations steps are borrowing-into and borrowing-from. When such a preparation subprocedure is taught, the learner sees a second setting for the main step, Diff. This new setting allows the learner to induce what is common about the two occurrences and abstract it into a satisfaction condition. The attachment of the new material is done with a satisfaction condition. In the case of <X Y Z M>, the resulting goal structure would be:

```
Goal: G    Satisfaction condition: SC
  1.  AC ⇒ P
  2.  true   ⇒ M

Goal: P    Satisfaction condition: False
  1.  X
  2.  Y
  3.  Z
```

where SC is some condition achieved by M, and AC is some condition indicating that the new preparation goal P is needed. This is a fine story in that it ties in well with the step schemata account of learning (see section 19.2).

However, it makes wrong predictions. Just as it predicts that regular borrowing will be learned this way, yielding the all important satisfaction condition of Sub1Col, it also predicts that borrowing from zero will be learned this way. Teleologically, borrowing is just as much a setup in one case as in the other. The textbooks even take pains to point this out. So the prediction is that the following structure would be required for BFZ:

Goal: Borrow-from (C) Satisfaction condition: C's top digit is decremented
1. T=0 in C ⇒ (BFZ C)
2. true ⇒ (Decrement-top C)

Goal: BFZ (C) Satisfaction condition: false
1. (Add10 C)
2. (Borrow-from (Next-column C))

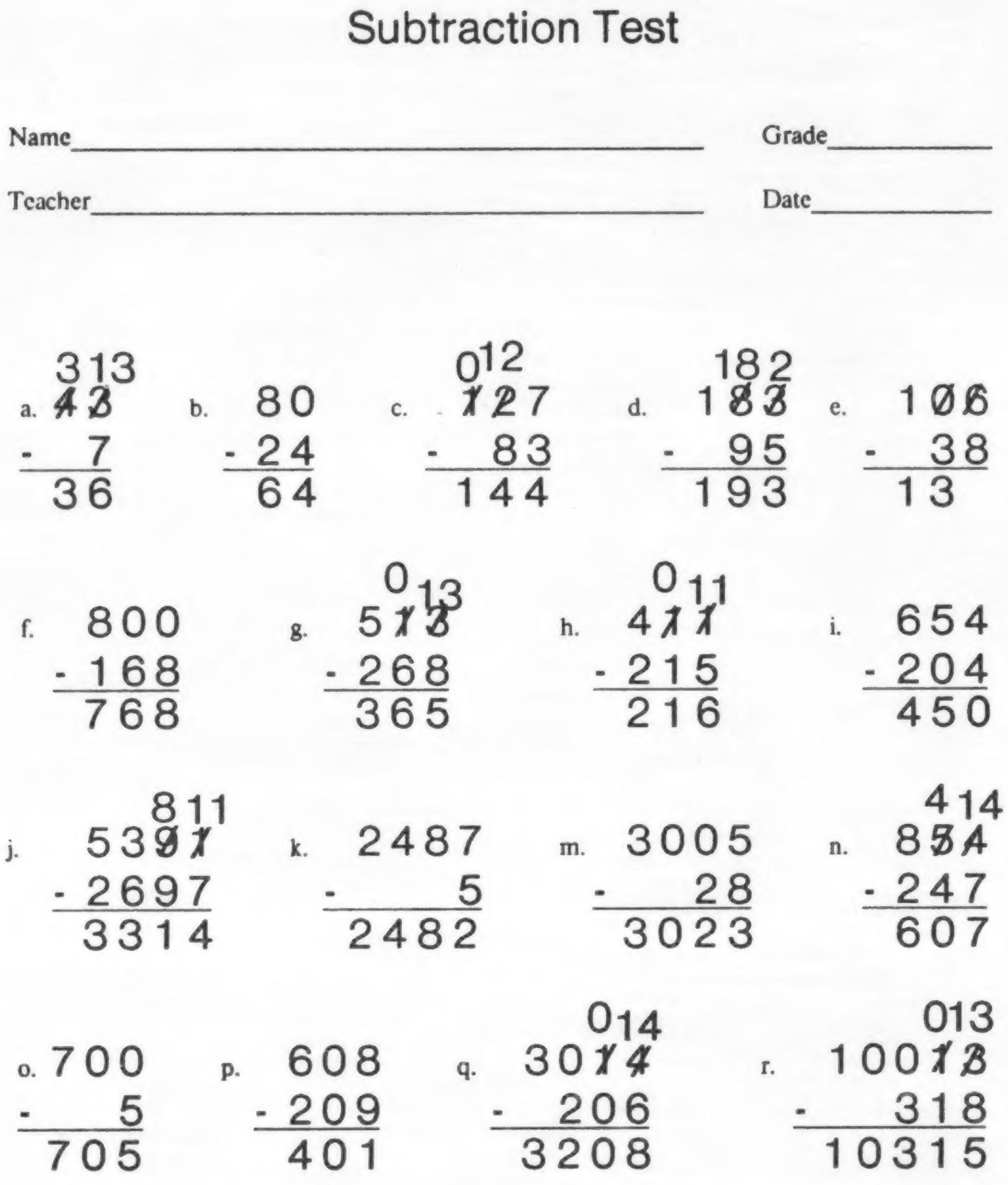
There is a satisfaction condition on the Borrow-from goal, and there is no decrement action under BFZ. Because the decrement action is been discovered to be a main step, it occurs only once, under Borrow-from. This structure is isomorphic to the structure used for borrowing, whose main step is Diff.

Given this decomposition, rule deletion can't generate the deletion bug Don't-Decrement-Zero. To do so, it needs to delete the decrement rule which occurs after BFZ's Add10 action (rule 18 in figure 10–1), but to leave the decrement rule of simple borrowing alone. Under goal structure given above, those two decrement rules are the same. Rule deletion can't delete just one. So, one of the three crucial deletion bugs can't be generated. If it is to be generated, then satisfaction condition acquisition must not induce a satisfaction condition for Borrow-from when it learns BFZ. It must instead use an OR-goal structure, similar to the one used in figure 10–1. But why is a satisfaction condition learned for Sub1Col but not for Borrow-from? If the acquisitional account that solves the blank answer critic problem is going to stand up, then it must explain why a satisfaction condition is acquired for one goal but not the other.

A10.3 Blank answer blocking: the acquisitional timing is wrong

The acquisitional account given above makes the prediction that the Sub1Col satisfaction condition will be acquired when simple, non-zero borrowing is learned. Hence, after students learned simple borrowing, they should no longer leave blanks in their answers. There is some data contradicting this prediction. Figure A10–1 reproduces a test taken by a third grader, student 3 of class 2. Except for three test items, the student answers as if he had a compound of three bugs (i.e., 0−N=N, Borrow-Once-Then-Smaller-From-Larger, and Smaller-From-Larger-Instead-of-Borrow-From-Zero). One of these three test items is the datum that is important here. It is the very first BFZ problem, problem *e*. From the scratch marks in the tens column, it is apparent that the student attempts to decrement the zero and hit an impasse. He apparently repairs using the Backup repair. He resumes execution by trying to process the units column and discovers that he can't because the column is still in its original T<B form. This is a second impasse. He repairs with the Noop repair, causing him to essentially give up on the units column and go on to the next column. The rest of the problem he answers in his usual way, which includes the bug 0−N=N (c.f., the tens column). His performance of the units column is characteristic of a bug called Blank-Instead-of-Borrow-From-Zero.

Problem *e* is the only evidence (so far) that this bug exists. However, the analysis of problem *e* is supported by this student's performance on all the other BFZ problems on the test. The other BFZ problems are answered with the bug Smaller-from-Larger-Instead-of-Borrow-From-Zero (problems *m* and *n* are evidence for the bug; problems *f*, *o* and *r* are answered by 0−N=N). This bug is generated by following the same course that the derivation of Blank-Instead-of-Borrow-From-Zero followed, except that the second impasse, the T<B impasse in the units column, is repaired by Refocus instead of Noop. That is, the two bugs are in the same bug migration class. They come from the same core procedure. It seems quite clear that this student knows about simple, non-zero borrowing. But the first time he encounters a BFZ problem, he impasses and repairs in a way characterized by Blank-Instead-of-Borrow-From-Zero. The other BFZ problems



Subtraction Test

Name________________________ Grade__________

Teacher________________________ Date__________

a. 313
$\cancel{4}\cancel{3}$
- 7
36

b. 80
- 24
64

c. 012
$\cancel{1}\cancel{2}7$
- 83
144

d. 182
$1\cancel{8}\cancel{3}$
- 95
193

e. $1\cancel{0}\cancel{6}$
- 38
13

f. 800
- 168
768

g. 0 13
$5\cancel{1}\cancel{3}$
- 268
365

h. 0 11
$4\cancel{1}\cancel{1}$
- 215
216

i. 654
- 204
450

j. 811
$53\cancel{9}\cancel{1}$
- 2697
3314

k. 2487
- 5
2482

m. 3005
- 28
3023

n. 414
$8\cancel{5}\cancel{4}$
- 247
607

o. 700
- 5
705

p. 608
- 209
401

q. 014
$30\cancel{1}\cancel{4}$
- 206
3208

r. 013
$100\cancel{1}\cancel{3}$
- 318
10315

Figure A10–1
A test showing Blank-Instead-of-Borrow-From-Zero

are repaired slightly differently. This is evidence that Blank-Instead-of-Borrow-From-Zero exists. Since there is only this one problem to support the existence of the bug, it could perhaps be dismissed as a fluke. Perhaps the student got rattled by the impasse and temporarily abandoned his knowledge of the procedure. But he is not so rattled that he fails to finish the problem, so it is equally plausible that this is a true bug migration.

The existence of this bug contradicts the conjecture that students will stop leaving blanks in the answer when they learn simple borrowing. Here is a student that has learned simple borrowing, and yet he leaves a blank in the answer. The prediction still stands that by the time students have learned BFZ, they will stop leaving blanks in the answer (c.f., the earlier discussion of *Blank-With-Borrow). However, it appears that linking the acquisition of blank-blocking with borrowing is a little premature. This casts doubt on the whole satisfaction condition framework for blocking the star bugs that generate blanks.

A10.4 Ill-formed answers

Two arguments were given against using satisfaction conditions to block the star bugs that generate gaps in answers. One was based on the fact that Sub1Col needs to have a satisfaction condition but Borrow-from needs not have one. Any account of how satisfaction conditions are acquired would have to explain why one goal and not the other acquires a satisfaction condition. The second argument indicated that the acquisition of the blank-blocking subskill might not occur at the time the appropriate satisfaction conditions are acquired.

A totally different approach to the blank answer critic problem is to focus on the notation rather than the procedure. The basic idea is that it is not the fact that Sub1Col wants to answer columns that prevents blanks, but the fact that answers must have a certain syntax, and that syntax excludes blanks in the middle of numbers. This solves the mystery of why some goals seem to acquire satisfaction conditions and others don't. Sub1Col seems to have a satisfaction condition because answer blanks are blocked by knowledge that they make the answer ill-formed. That is, *Blank-with-Borrow is blocked because it produces syntactically ill-formed notation. On the other hand, Borrow-from seems not to have a satisfaction condition because there is nothing ill-formed about a column that lacks a decrement. Hence, the bug Don't-Decrement-Zero is not blocked, because it generates only syntactically correct notation. So the general idea is that no goal has a satisfaction condition. What appeared to be satisfaction conditions was just syntactic knowledge being applied somehow to block bugs.

Using knowledge of notational syntax to block the star bugs also solves the mystery involving the timing of acquisition. The acquisition of knowledge about the notation would be decoupled from the acquisition of the procedure *per se*. Hence, there would be nothing unusual about a student, such as the third grader mentioned above, who knew how to do non-zero borrowing but did not know to filter repairs that leave blanks in the answer. The acquisition of borrowing apparently occurred before the knowledge of the ill-formedness of gapped answers.

The presence of such notational knowledge is much clearer in algebra than in subtraction. It's a widely accepted empirical generalization that algebra students almost always produce syntactically well-formed answers. The answers might be wrong, but they are syntactically correct. Carry et al. (1978) present hundreds of error types, and all are syntactically well formed. Indeed, Carry et al. assume that students impose syntactical well-formedness on their answers in order to explain several classes of errors. For instance, they propose a general deletion transformation that excises subexpressions from algebraic expressions. A common example involves cancelling, as in

$$\frac{x}{3+x} \Rightarrow \frac{1}{3}$$

The two instances of the variable have been cancelled. Carry et al. note that if such deletions were taken literaly, they would leave syntactically malformed expressions:

$$\frac{x}{3+x} \Rightarrow \frac{\quad}{3+}$$

Subjects don't do this. They fill in blanks with zero or one, and they delete extra operator signs. Apparently, they do this in order to make the output expression syntactically well formed. This adherence to well-formedness applies to intermediate expressions, as well as to the final expression. In particular, when the task is to solve an equation, one sees line after line of well-formed equations produced. Apparently, students will repair syntactic malformations of intermediate expressions before going on to the next transformation. They don't wait until the end to check the syntax. The same is true of blanks in subtraction answers. The students don't wait until the end to repair a blank, they fill it right away.

This solution to the problem of generating star bugs is not well formulated yet. In particular, nothing implementing it exists in Sierra at this time. However, it is a more promising direction than using satisfaction conditions to achieve the same blocking.

Taking the syntactic approach removes the motivation for satisfaction conditions. There is no reason to use the more powerful goal type now. To keep satisfaction conditions in the representation anyway is possible, but creates the problem that the learner must be equipped to learn satisfaction conditions. As indicated earlier, satisfaction condition acquisition is problematic. Since there is no motivation for satisfaction conditions and their presence in the representation creates extra difficulties for the learning theory, the satisfaction condition position will be abandoned. The goal type convention for the representation will be the And-Or convention, which is simpler and weaker than the satisfaction conditions anyway.

Robert Ahlers
Code N711
Human Factors Laboratory
NAVTRAEQUIPCEN
Orlando, FL 32813

Liaison Scientist
Office of Naval Research
Branch Office, London
Box 39
FPO New York, NY 09510

Dr. Richard Cantone
Navy Research Laboratory
Code 7510
Washington, DC 20375

Chief of Naval Education and Training
Liaison Office
Air Force Human Resource Lab.
Operations Training Division
WILLIAMS AFB, AZ 85224

Dr. Stanley Collyer
Office of Naval Technology
800 N. Quincy Street
Arlington, VA 22217

CDR Mike Curran
Office of Naval Research
800 N. Quincy Street
Arlington, VA 22217

Dr. Meryl S. Baler
Navy Personnel R&D Center
San Diego, CA 92152

Dr. Pat Federico
Code P13
NPRDC
San Diego, CA 92152

Dr. John Ford
Navy Personnel R&D Center
San Diego, CA 92152

Dr. Jude Franklin
Code 7510
Navy Research Laboratory
Washington, DC 20375

Dr. Mike Gaynor
Navy Research Laboratory
Code 7510
Washington, DC 20375

Dr. Jim Hollan
Code 304
Navy Personnel R&D Center
San Diego, CA 92152

Dr. Ed Hutchins
Navy Personnel R&D Center
San Diego, CA 92152

Dr. Joe McLachlan
Navy Personnel R&D Center
San Diego, CA 92152

Dr. Norman J. Kerr
Chief of Naval Technical Training
Naval Air Station Memphis (75)
Millington, TN 38054

Dr. Peter Kincaid
Training Analysis & Evaluation Group
Dept. of the Navy
Orlando, FL 32813

Dr. James Lester
ONR Detachment
495 Summer Street
Boston MA 02210

Dr. William L. Maloy (02)
Chief of Naval Education and Training
Naval Air Station
Pensacola, FL 32508

Dr. William Montague
NPRDC Code 13
San Diego, CA 92152

Library, Code P201L
Navy Personnel R&D Center
San Diego, CA 92152

Technical Director
Navy Personnel R&D Center
San Diego, CA 92152

Commanding Officer
Naval Research Laboratory
Code 2627
Washington, DC 20390

Office of Naval Research
Code 433
800 Quincy Street
Arlington, VA 22217

Personnel & Training Research Group
Code 442PT
Office of Naval Research
Arlington, VA 22217

Office of Chief of Naval Operations
Research Development&Studies
OP 115
Washington, DC 20350

Dr. Gil Ricard
Code N711
NTEC
Orlando, FL 32813

Dr. Worth Scanland
CNET (N-5)
NAS, Pensacola, FL 32508

Dr. Robert G. Smith
Office of Chief of Naval Operations
OP-987H
Washington, DC 20350

Dr. Alfred F. Smode, Director
Training Analysis & Evaluation Group
Dept. of the Navy
Orlando, FL 32813

Dr. Richard Sorensen
Navy Personnel R&D Center
San Diego, CA 92152

Dr. Frederick Steinheiser
CNO - OP115
Navy Annex
Arlington, VA 20370

Roger Weissinger-Baylon
Department of Administrative Sciences
Naval Postgraduate School
Monterey, CA 93940

Mr. John H. Wolfe
Navy Personnel R&D Center
San Diego, CA 92152

H. William Greenup
Education Advisor (E031)
Education Center, MCDEC
Quantico, VA 22134

Special Assistant /M.C. Matters
Code 100M
Office of Naval Research
800 N. Quincy Street
Arlington, VA 22217

Dr. A.L. Slafkosky
Scientific Advisor (code RD-1)
HQ, U.S. Marine Corps
Washington, DC 20380

Technical Director
U.S. Army Research Intitute for
Behavorial and Social Sciences
5001 Eisenhower Avenue
Alexandria, VA 22333

Mr. James L.Baker
Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

Dr. Beatrice J. Farr
U.S. Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

Dr. Milton S. Katz
Training Technical Area
U.S. Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

Dr. Marshall Narva
U.S. Army Research Intitute for
Behavorial and Social Sciences
5001 Eisenhower Avenue
Alexandria, VA 22333

Dr. Harold F. O'Neil, Jr.
Director, Training Research Lab
Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

Commander, U.S. Army Research
Institute for the Beh. & Soc. Sciences
ATTN: PERI-BR (Dr. Judith Orasanu)
5001 Eisenhower Av.
Alexandria, VA 22333

Joseph Psotka, Ph.D.
ATTN: PERI-1C
Army Research Institute
5001 Eisenhower Av.
Alexandria, VA 22333

Dr. Robert Wisher
Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

Dr. Robert Sasmor
U.S. Army Research Intitute for
Behavorial and Social Sciences
5001 Eisenhower Avenue
Alexandria, VA 22333

Technical Documents Center
Air Force Human Resources
Laboratory
WPAFB, OH 45433

U.S. Air Force Office of Scientific
Research
Life Sciences Directorate, NL
Bolling Air Force Base
Washington, DC 20332

Air University Library
AUL/LSE 76/443
Maxwell AFB, AL 36112

Bryan Dallman
AFHRL/LRT
Lowry AFB, CO 80230

Dr. Genevieve Haddad
Program Manager
Life Sciences Directorate
AFOSR
Bolling AFB, DC 20332

Dr. T.M. Longridge
AFHRL/OTGT
Williams AFB, AZ 85224

Dr. Joseph Yasatuke
AFHRL/OTGT
Williams AFB, AZ 58224

Defense Technical Information Center
Cameron Station, Bldg 5
Alexandria, VA 22314
Attn: TC

Military Assist., Training and
Personnel Technology
Office of Under Secretary of Defense
Room 3D129, The Pentagon
Washington, DC 20301

Major Jack Thorpe
DARPA
1400 Wilson Blvd.
Arlington, VA 22209

Dr. Susan Chipman
Learning and Development
National Institute of Education
1200 19th Street NW
Washington, DC 20208

Edward Esty
Dept. of Education, OERI
MS 40
1200 19th St., NW
Washington, DC 20208

Dr. John Mays
National Institute of Education
1200 19th Street NW
Washington, DC 20208

Dr. Arthut Melmed
OERI
1200 19th Street NW
Washington, DC 20208

Chief, Psychological Research Branch
U.S. Coast Guard (G-P-1/2/TP42)
Washington, DC 20593

Dr. Edward C. Weiss
National Science Foundation
1800 G. Street, NW
Washington, DC 20550

Dr. Frank Withrow
U.S. Office of Education
400 Maryland Avenue, SW
Washington, DC 20202

Dr. Joseph L. Young, Director
Memory & Cognitive Processes
National Science Foundation
Washington, DC 20550

Dr. John R. Anderson
Department of Psychology
Carnegie-Mellon University
Pittsburgh, PA 15213

Dr. Michael Atwood
Bell Laboratories
11900 North Pecos St.
Denver, CO 80234

Psychological Research Unit
Dept. of Defense (Army Office)
Campbell Park Offices
Canberra ACT 2600
AUSTRALIA

Dr. Patricia Baggett
Department of Psychology
University of Colorado
Boulder, CO 80309

Mr. Avron Barr
Department of Computer Science
Stanford University
Stanford, CA 94305

Dr. John Black
Yale University
Box 11A, Yale Station
New Haven, CT 06520

Dr. Glen Bryan
6208 Poe Road
Bethesda, MD 20817

Dr. Bruce Buchanan
Deparment of Computer Science
Stanford University
Stanford, CA 94305

Dr. Jaime Carbonell
Carnegie-Mellon University
Department of Psychology
Pittsburgh, PA 15213

Dr. Pat Carpenter
Department of Psychology
Carnegie-Mellon University
Pittsburgh, PA 15213

Dr. William Chase
Department of Psychology
Carnegie-Mellon University
Pittsburgh, PA 15213

Dr. Micheline Chi
Learning R & D Center
University of Pittsburgh
3939 O'Hara Street
Pittsburgh, PA 15213

Dr. William Clancey
Department of Computer Science
Stanford University
Stanford, CA 94306

Dr. Michael Cole
UCSD
Lab. of Comparative Human
Cognition - D0034
La Jolla, CA 92093

Dr. Allan M. Collins
Bolt Beranek & Newman, Inc.
50 Moulton Street
Cambridge, MA 02138

ERIC Facility-Acquisitions
4833 Rugby Avenue
Bethesda. MD 20014

Dr. Paul Feltovich
Medical Education Department
SIU School of Medicine
P.O. Box 3926
Springfield, IL 62708

Mr. Wallace Feurzeig
Educational Technology Department
Bolt Beranek & Newman
10 Moulton St.
Cambridge, MA 02238

Dr. Dexter Fletcher
WICAT Research Institute
1875 S. State St.
Orem, UT 22333

Dr. John R. Frederiksen
Bolt Beranek & Newman
50 Moulton Street
Cambridge, MA 02138

Dr. Michael Genesereth
Department of Computer Science
Stanford University
Stanford, CA 94305

Dr. Don Gentner
Center for Human Information
Processing
UCSD
La Jolla, Ca 92093

Dr. Dedre Gentner
Bolt Beranek & Newman
10 Moulton St.
Cambridge MA 02138

Dr. Robert Glaser
Learning R&D Center
University of Pittsburgh
3939 O'Hara Street
Pittsburgh, PA 15260

Dr. Marvin D. Glock
217 Stone Hall
Cornell University
Ithaca, NY 14583

Dr. Joseph Goguen
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025

Dr. Daniel Gopher
Department of Psychology
University of Illinois
Champaign, IL 61820

Dr. James G. Greeno
LRDC
University of Pittsburgh
3939 O'Hara Street
Pittsburgh, PA 15213

Dr. Barbara Hayes-Roth
Department of Computer Science
Stanford University
Stanford, CA 95305

Dr. Frederick Hayes-Roth
Teknowledge
525 University Ave.
Palo Alto, CA 94301

Dr. James R. Hoffman
Department of Psychology
University of Delaware
Newark, DE 19711

Glenda Greenwald, Ed.
Human Intelligence Newsletter
P. O. Box 1163
Birmingham, MI 48012

Dr. Earl Hunt
Department of Psychology
University of Washington
Seattle, WA 98105

Dr. Marcel Just
Dept. of Psychology
Carnegie-Mellon University
Pittsburgh, PA 15213

Dr. Scott Kelso
Haskins Laboratories, Inc.
270 Crown Street
New Haven, CT 06510

Dr. David Kieras
Department of Psychology
University of Arizona
Tuscon, AZ 85712

Dr. Walter Kintsch
Department of Psychology
University of Colorado
Boulder, CO 80302

Dr. Stephen Kosslyn
Department of Psychology
Brandeis University
Waltham, MA 02254

Dr. Pat Langley
Carnegie Mellon University
Pittsburgh, PA 15260

Dr. Jill Larkin
Department of Psychology
Carnegie Mellon University
Pittsburgh, PA 15213

Dr. Alan Lesgold
LRDC
University of Pittsburgh
3939 O'Hara St.
Pittsburgh, PA 15260

Dr. Jim Levin
UCSD
Laboratory for Comparative Human
Cognition--D003A
La Jolla, CA 92093

Dr. Michael Levine
Department of Educational Psycholo
210 Education Building
University of Illinois
Champaign, IL 61801

Dr. Jay McClelland
Department of Psychology
MIT
Cambridge, MA 02139

Dr. James R. Miller
Computer Thought Corp.
1721 West Plano Highway
Plano, TX 75075

Dr. Mark Miller
Computer Thought Corporation
1721 West Plane Parkway
Plano, TX 75075

Dr. Tom Moran
Xerox PARC
3333 Coyote Hill Road
Palo Alto, CA 94304

Dr. Allen Munro
Behavioral Technology Laboratories
1845 Elena Ave., Fourth Floor
Redondo Beach, CA 90277

Dr. Donald A. Norman
Cognitive Science, C-015
UCSD
La Jolla, CA 92093

Dr. Seymour A. Papert
MIT
Artifical Intelligence Lab.
545 Technology Square
Cambridge, MA 02139

Dr. Nancy Pennington
University of Chicago
5801 S. Ellis Avenue
Chicago, IL 600637

Dr. Peter Polson
Department of Psychology
University of Colorado
Boulder, CO 80309

Dr. Fred Reif
Physics Department
University of California
Berkeley, CA 94720

Dr. Lauren Resnick
LRDC
University of Pittsburgh
3939 O'Hara Street
Pittsburgh, PA 15260

Mary S. Riley
Program in Cognitive Science
Center of Human Info. Processing
UCSD
La Jolla, CA 92093

Dr. Ernst Z. Rothkopf
Bell Laboratories
Murray Hill, NJ 07974

Dr. William B. Rouse
Georgia Institute of Technology
School of Industrial & Systems Engineering
Atlanta, GA 30332

Dr. David Rumelhart
Center for Human Information Processing
UCSD
La Jolla, CA 92093

Dr. Michael J. Samet
Perceptronics, Inc.
6271 Variel Avenue
Woodland Hills, CA 91364

Dr. Arthur Samuel
Yale University
Department of Psychology
Box 11A, Yale Station
New Haven, CT 06520

Dr. Roger Schank
Yale University
Department of Computer Science
P.O. Box 2158
New Haven, CT 06520

Dr. Walter Schneider
Department of Psychology
603 E. Daneil
Champaign, IL 61

Dr. Alan Schoenfeld
Mathematics and Education
University of Rochester
Rochester, NY 14627

Mr. Colin Sheppard
Applied Psychology Unit
Admiralty Marine Technology Est.
Teddington, Middlesex
United Kingdom

Dr. H.W. Sinaiko, Program Director
Manpower Research&Advisory Service
Smithsonian Institution
801 North Pitt Street
Alexandria, VA 22314

Dr. Edward E. Smith
Bolt Beranek & Newman, Inc.
50 Moulton Street
Cambridge, MA 02138

Dr. Eliott Soloway
Yale University
Department of Computer Science
P.O. Box 2158

Dr. Kathryn T. Spoehr
Psychology Department
Brown University
Providence, RI 02812

Dr. Robert Sternberg
Psychology Department
Yale University
Box 11A, Yale Station
New Haven, CT 06520

Dr. Albert Stevens
Bolt Beranek & Newman, Inc.
10 Moulton St.
Cambridge, MA 02238

David E. Stone, Ph.D.
Hazeltine Corporation
7680 Old Springhouse Road
McLean, VA 22102

Dr. Patrick Suppes
Institute for Mathematical Studies in
Social Sciences
Stanford University
Stanford, CA 94305

Dr. Kikumi Tatsuoka
Computer Based Education
Research Lab
252 Engineering Research Lab
Urbana, IL 61801

Dr. Maurice Tatsuoka
220 Education Bldg.
1310 S. Sixth St.
Champaign, IL 61820

Dr. Perry W. Thorndyke
Perceptronics, Inc.
245 Park Lane
Atherton, CA 94025

Dr. Douglas Towne
USC
Behavorial Technology Labs
1845 S. Elena Ave.
Redondo Beach, CA 90277

Dr. Keith T. Wescourt
Perceptronics, Inc.
545 Middlefield Road, Suite 140
Menlo Park, CA 94025

Dr. Mike Williams
XEROX PARC
3333 Coyote Hill Road
Palo Alto, CA 94304